Universitat de Lleida

TREBALL FINAL DE MÀSTER

ESCOLA
POLITÈCNICA SUPERIOR
UNIVERSITAT DE LLEIDA
INSPIRING THE FUTURE

Estudiant: Gerard Rovira Sánchez

Titulació: Màster en Enginyeria Informàtica

Títol de Treball Final de Màster: Verification of Self-Sovereign Identities in Ethereum applied to a Media Reuse Smart Contracts Scenario

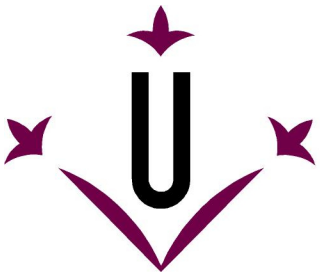
Director/a: Juan Manuel Gimeno Illa, Roberto García González

Presentació

Mes: Març

Any: 2019

Verification of Self-Sovereign Identities in Ethereum applied to a Media Reuse Smart Contracts Scenario



Gerard Rovira Sánchez

Polytechnic School
University of Lleida

A Thesis presented for the degree of
Master in Computer Science

Advisors:
Juan Manuel Gimeno Illa
Roberto García González

March 1, 2019

Abstract

As a decentralized blockchain network, Ethereum enables us to do immutable, tamper-proof and secure transactions. However, its current design makes it very difficult to trace the real owner behind an address, if not impossible.

This thesis aims to give a solution to the verification of identities behind Ethereum addresses, as well as to demonstrate how a third party service can take advantage of it.

In particular, we have worked with InVID Rights Management web platform to provide a blockchain-based service to guarantee that the rights given over social media videos are safe, transparent and non-repudiable.

Acknowledgements

I would first like to thank my thesis advisors Juan Manuel Gimeno Illa and Roberto García González. They have both been accessible when I needed it, no matter the day of the week, and they have given me excellent support throughout the research, implementation, and documentation of the project.

Prior to starting the project, I just had a vague idea about what Ethereum and IPFS were about. Juan Manuel and Roberto guided me to get myself started with decentralized technology rapidly and make sure I was on the right track at all times. They were also the ones who spotted the Verification of Identities issue in the first place, and the opportunity to build a solution on top of Ethereum.

I would also like to thank Albert Berga Gatius, the main developer of the InVID Rights Management API, for his help throughout the development of the project, and enabling blockchain on InVID Rights Management server-side system.

I would also like to thank InVID, for giving me the opportunity to work on a blockchain solution to handle reuse requests on this last deliverable before the presentation of the project to the EU commission.

Finally, my sincere gratitude also goes to my parents and sister for their continuous support and encouragement throughout my years of study and also during the writing of this thesis.

Thank you.

Contents

List of Figures

Chapter 1

Introduction

1.1 Overview

Blockchain utilizes decentralized technology to create various kinds of networks. It is the underlying technology for digital currencies like Bitcoin or Ethereum and can be used for many different purposes, such as a system of record for digital identities[3], tokenization or voting.

The data is distributed, verified and recorded on public ledgers. A distributed ledger can be seen as a network of personal computers, a system which nobody fully owns[4]. The networks are trustless, and nodes store replicas of the data and synchronize themselves according to a well-known consensus. In a distributed ledger system there is no administrator or centralized data storage.

The well-known consensus protocols are defined by each of the networks. The two most popular protocols are Proof of Work (PoW) and Proof of Stake (PoS). Nodes verify and confirm new data by using these protocols.

In a blockchain network, the data is stored in blocks, and for a successful block to be added in the network, the new block is validated against a Merkle Tree [1] of previous network blocks. The addition of new blocks into the network forms the chain of blocks. Blocks encapsulate transactions, sent by network nodes, and block creators are responsible for creating these blocks of transactions according to the consensus.

Proof of Work is the first consensus protocol created for Bitcoin. It is also being used by Ethereum as of late 2018. The aim of PoW is to make it difficult for miners to generate new blocks. Block creators (known as miners) have to run a hashing

[1] `https://bitcoin.org/en/glossary/merkle-tree`

function to determine the next valid block. They are granted a reward for each valid block pushed to the network. PoW difficulty, defined in the consensus, regulates the flux of blocks in the network. In a PoW network the longest chain wins, so nodes will trust the longest chain discovered in the network.

Proof of Stake is an increasingly popular consensus protocol, first used by Peercoin. Ethereum is currently working on a hybrid PoW-PoS implementation under the name Casper. PoS It aims to be as safe (if not safer) than PoW while being more energy efficient. With PoS there is no hashing function to resolve and the creator of new blocks is chosen in a deterministic way based on their stake (the coin or tokens they possess)[4]. Contrary to PoW there is no reward given to block creators. Instead, they take transaction fees.

Blockchain decentralization makes it very difficult to hack or corrupt since attacks have to focus on the whole network, rather than a single node. Hacking a network node would have no impact on the network whatsoever, and to change the course of the chain of blocks the attacker would need at least 51% of the hashing power in the network[5].

Nonetheless, decentralized networks compared to similar centralized solutions are still at the very early stages. Currently, both Bitcoin and Ethereum present some issues that should be addressed sooner or later:

High computing cost

Currently, both platforms are based on PoW (Proof of Work), which requires a high computational cost to calculate the following blocks on the chain. PoS (Proof of Stake) aims to give a solution to that by giving control to the most trustworthy nodes in the network at random, but neither Bitcoin nor Ethereum has a working solution yet.

Size of the blockchain

The current implementations make it mandatory for each of the network nodes to fetch every single block, including its transactions and metadata.

While this makes the network more robust, since every node will verify that the next block is correct according to the Merkle-tree hashing, it also makes it difficult to scale. The Ethereum blockchain takes slightly over 1TB as of 2018[6] but may take double that in the next 1-2 years. Syncing a full node also takes many hours now.

Just like it has been done with databases and filestores, blockchain enthusiasts suggest sharding[7] as a way to address this problem. However, a sharding solution

would come with compromises: the networks would be more prone to 51% attacks because the hashing power would be distributed across the number of shards in the network.

Trusted parties

Bitcoin and Ethereum addresses are self-generated, from a self-generated random 64 hexadecimal characters private key. The public key and public address are then derived from the private key by using the Elliptic Curve Digital Signature Algorithm (ECDSA)[8].

That enables individuals to generate themselves one or many addresses, which immediately enables them to send transactions all over the network.

For Bitcoin or Ethereum these identities are pseudo-anonymous. The addresses themselves are random and the transactions from these accounts are only identified by their account address. Hence, the transactions seem completely anonymous. However, their current design makes it possible to trace in/out transactions from these addresses, which would eventually be enough to prove the identity the real owner behind the account. Also, it is not possible to transact without Ether, which makes it more difficult to be anonymous on the network.

Not all blockchain implementations are pseudo-anonymous. Monero or ZCash are two examples of cryptocurrencies which focus on anonymity[9].

Either way, it is not easy to discover the real identity behind a blockchain account, even if it is not within the owner intentions to be anonymous. For example, an online store willing to send an email notification to a customer after they have done a purchase on their store has no means (through blockchain technology only) to retrieve their rightful email. On the other hand, when authenticating through Facebook or Google, this information is already verified by the corresponding authentication service provider.

The verification of identities with Ethereum is the topic we will be focusing on in this MSc thesis.

Other than the three listed issues, the Ethereum wiki presents a more in-depth list[2] of the current problems that they are facing, most of them apply to other blockchains too.

[2]Ethereum problems: `https://github.com/ethereum/wiki/wiki/Problems`

1.2 Motivation

The verification of identities is critical for certain platforms such as InVID Rights Management[3]. As part of their development team, our project will consist of both the implementation of the identity verification application and its integration with the InVID project.

The InVID platform, as a whole, enables journalists to stay up to date with the latest trends, verify social media videos to detect fakes or duplicates and request reuse permission over them.

The InVID Rights Management module takes care of this last feature, on which journalists can currently submit a video from either Facebook, Twitter or YouTube and request reuse permission to the original owner. If the content owner, verified through each of the platform's API, agrees over the requested rights, a *contract* is signed.

However, the virtual contract should ideally have the same binding power as a regular one. That is more complicated and costly with a traditional centralized means (own client, server and database) because they are all controlled by the same party, or a small set of them, and thus this approach requires a trusted third party.

For instance, in the context of the InVID project the trust issues are the following:

- The database is solely under InVID's control. Content owners can argue that the database can be tampered by InVID personnel.
- Given that the web runs on the project's server and it is based on private source code, it is difficult to demonstrate how secure or reliable the system is.
- The project might not live forever, but agreements between journalists and content owners should persist even if the platform is down or dead.

For these reasons, we consider a decentralized blockchain solution to fit InVID Rights Management, and we will work on the identity verification over blockchain to make it easier for InVID, or for any other decentralized application, to identify who really is behind an address. Of course, this should not be at the cost of privacy. Their identity should only be disclosed if the account owner is willing to.

Our project will be based on Ethereum, one of the most robust and powerful decentralized platforms. What makes Ethereum the clear choice for this project is the ability to create what they call *smart contracts*. Smart contracts are pieces

[3]InVID Rights Management: `https://rights.invid.udl.cat`

of code written mainly using the Solidity programming language that anyone can write and upload to the Ethereum network. Every single network node will validate and execute the given code (through the Ethereum Virtual Machine), data can be read by each of the network nodes and smart contract methods can be executed via transactions.

Smart contracts enable developers to carry credible complex transactions without the need for third parties.

Additionally, we will be exploring uPort. It works on top of Ethereum to provide a solution to managed identities. uPort is maintaining a mobile application for users to control their decentralized identity that also works as a blockchain wallet, and an extensive API to interact with it and Ethereum.

1.3 Concept

The proposed solution is inspired by both the traditional email verification process and the Crypto Valley[4] initiative. In this initiative, Switzerland citizens have the possibility to acquire a digital identity on the Ethereum blockchain linked to their official government data[10]. Such a system involves matching public decentralized keys with personal data such as name, last name or birth date.

Both proposed systems have something in common, they rely on a point of trust. The point of trust is an individual, group of people or entity whom they can trust the validation. For the previous Switzerland example, we have to trust that the government has properly verified the name, last name and birthday of each of the citizens.

We will create an external service, uSocial (named after uPort), that will act as this point of trust, storing the various verifications on the Ethereum network by means of smart contracts. Platforms of any kind, such as InVID Rights Management, will be able to request such data to perform their app-specific operations in a smooth and decentralized way.

The uSocial platform will be open source and aims to be managed by a reputable corporation which everyone can trust at some point.

Decentralization purists might argue that the adding point(s) of trust makes it centralized once again. In a way they are correct, only a/few trusted parties will be verifying the data instead of every single node in the Ethereum network. But,

[4]https://cryptovalley.swiss

to our knowledge, there is no viable way to develop such as a system cheat-proof. Moreover, none of the well-known blockchain platforms are fully decentralized. For example, the Ethereum infrastructure is decentralized, every single node is trustless, but the logic running behind each of the nodes is carefully reviewed and agreed upon by a small set of people running the Ethereum project.

By using uSocial, InVID Rights Management application will be fully decentralized. uSocial will take the responsibility for correctly matching blockchain address <-> email to link between Ethereum addresses and real identities, that InVID needs to check whether a user is the rightful owner of a video. and submit rights requests on the Ethereum network whom both verified parties have to sign.

The remaining InVID Rights Management validation can be done through Ethereum smart contracts.

1.4 Objectives

The scope of our project is the following:

Design and create the application to verify identities

First and foremost, we have to work on uSocial, the project that will enable the verification of credentials.

This project, built from the ground up, will be deployed as an isolated service.

We aim to build an easy-to-use platform, that enables the following verifications based on Ethereum:

- Facebook
- Google
- Twitter
- Email

Users should be able to verify one or more accounts for each of the platforms, as they can for instance have more than one email address or Google account.

Additionally, uPort users should be able to use account and verification tools through QR codes that can be scanned with the uPort mobile application, the only supported way, for now, to transfer information in/out the device. QR codes are a good mechanism though, they make is fast to pass long arbitrary characters and they prevent mistypes on critical information such as addresses and tokens.

Integrate uSocial with InVID Rights Management

When uSocial is ready and deployed, we will use it from the InVID Rights Management application as our point of trust. To secure a reuse request through blockchain, both journalists and content owners will have to have a registered uPort account, and in order for InVID to consider such account, it will require them to have at least an email verification.

When the journalist creates a reuse request through blockchain, it will be created in both InVID API and on the Ethereum network through smart contracts. The smart contracts will make sure the reuse request is only set as accepted/agreed once both parties have signed: they both have created a transaction stating they want to sign a given reuse request. The reuse request will be complete when both parties have saved their agreements in both the centralized InVID server and on the Ethereum smart contract.

1.5 Document Structure

The rest of this document is structured as follows:

In this section, we have briefly introduced blockchain, Ethereum, uPort as well as other related decentralized technology which may be interesting for the implementation of our project. We have also described the motivation behind it and why the Verification of Identities is so important in the context of this project.

In the State of the Art section, we will analyze Ethereum and uPort in more depth, and we will introduce concepts such as IPFS and Truffle which are going to be very useful throughout the project.

The State of Art will include how the web infrastructure and deployment will work since both uSocial and InVID Rights Management are web-based.

The Development section will mostly present design decisions and how we overcame the various difficulties that we faced throughout the development of the project. We will first explain the methodology that we used, and we will use the rest of the sections to focus on the development of the project.

The Development section will start with uSocial implementation, but will also include everything with regards to blockchain in InVID.

This document will end with a set of conclusions based on the development experience and results and a list of recommendations, features, and enhancements for future versions of the project.

Chapter 2

State of the Art

The State of the Art comprises technology, documentation and related work with the development of the uSocial platform and its integration with InVID Rights Management.

Therefore, it is organized in the following subsections: Ethereum, Web Architecture (Client and Server), and Continuous Deployment.

In any case, we will be putting emphasis on everything related to blockchain since it plays a key role in the Verification of Identities.

2.1 Ethereum

Regarding blockchain technologies, we will be working with Ethereum decentralized public blockchain network.

On the one hand, it is a tradable currency. *Ether*[1] is the name of the digital asset. There is also a second currency, *gas*, that it is used to pay transactions fees and the execution of smart contracts.

On the other hand, Ethereum makes it possible to execute arbitrary code on the blockchain through smart contracts. Smart contracts make it possible to store data on the blockchain and create currency transactions based on user-defined conditions. For example, set timers, ownership, tokens, etc.

The Ethereum Virtual Machine is Turing Complete and enables the execution of any program regardless of their size and the resources needed for its execution. The Ethereum platform created Solidity as the primary high-level language to write

[1]Small Ether values are often described in terms of Wei: $10^{18} Wei = 1 Ether$

smart contracts. To prevent nodes from abusing network nodes storage and CPU power, that can happen if the smart contract is too complex or eventually gets stuck in a loop, transaction owners have to add some extra gas to their transactions to pay for the cost of the smart contract storage they will use and CPU cycles of the functions they are executing. When considerable storage is needed, using IPFS (explained below) as a data storage is often a preferred choice as the smart contracts can get quite costly.

uPort, which we will get into more details next, uses a smart contract to store user identities and claims[11].

We will be using smart contracts to provide journalists a secondary and decentralized way to store media reuse requests in the context of InVID Rights Management. Specifically, the InVID Rights Management smart contract will manage:

- Involved parties: journalist and content owner identifiers.
- Reuse request metadata: identifier, revocation status and timestamps. For the second one, we will not deal with them directly. Instead, Solidity will do most of the work through built-in events.
- Reuse request steps metadata. In a perfect world, content owners would always agree on the requested rights over their media without presenting any objections. However, the reality is that sometimes they might not agree with some of the terms and conditions and they will want to propose new, more restrictive, ones. The stored metadata for each of the reuse request negotiation steps will contain the identifier of the step and whether the journalist and the content owner have signed/agreed on its terms.

The smart contract will make sure that only the appropriate individuals can sign the given steps, and that the terms cannot be modified. A reuse request can have infinite steps, but each of them should be immutable. Such immutability guarantees that both parties know exactly what they are agreeing on.

To store the content of the reuse request and the steps that they will be signing we will be using IPFS. Both can be quite lengthy JSON files, detailing each of the terms of the reuse request and their conditions, and we want to avoid lengthy content on our smart contract to lower transaction costs. Smart contracts were not designed to store big amounts of data.

IPFS, just like Ethereum, is also a decentralized solution, but this one focuses on data storage.

On IPFS, uploaded files are identified by their content fingerprint (also known as

cryptographic hash). This makes it impossible to modify a file without changing its fingerprint. Nodes can choose which files to distribute, contrary to Ethereum on which every single node has a copy of the whole content of the network. When searching for a file, the network is queried for a specific fingerprint; if many nodes are storing it the download can be executed in parallel from multiple nodes at once.

The way that IPFS can work with Ethereum is the following: we know that fingerprints are unique depending on the file content, and from the fingerprint, we can retrieve the whole file content back. We will move the big chunks of data that we want to store from Ethereum to IPFS and just store the fingerprint reference in Ethereum. To retrieve the full data, first the smart contract and then the linked IPFS fingerprints will have to be accessed.

For InVID, the strategy, shown in figure 2.1, will be the same. Our smart contract will save references to IPFS fingerprints to the complete data, for example, to store the full contents of the reuse request terms and conditions. Parties signing each of the reuse request steps are supposed to read through the IPFS content beforehand. Specifically, to retrieve IPFS data through fingerprints we will be using the Infura network[2], so we will not need to deploy our own IPFS node.

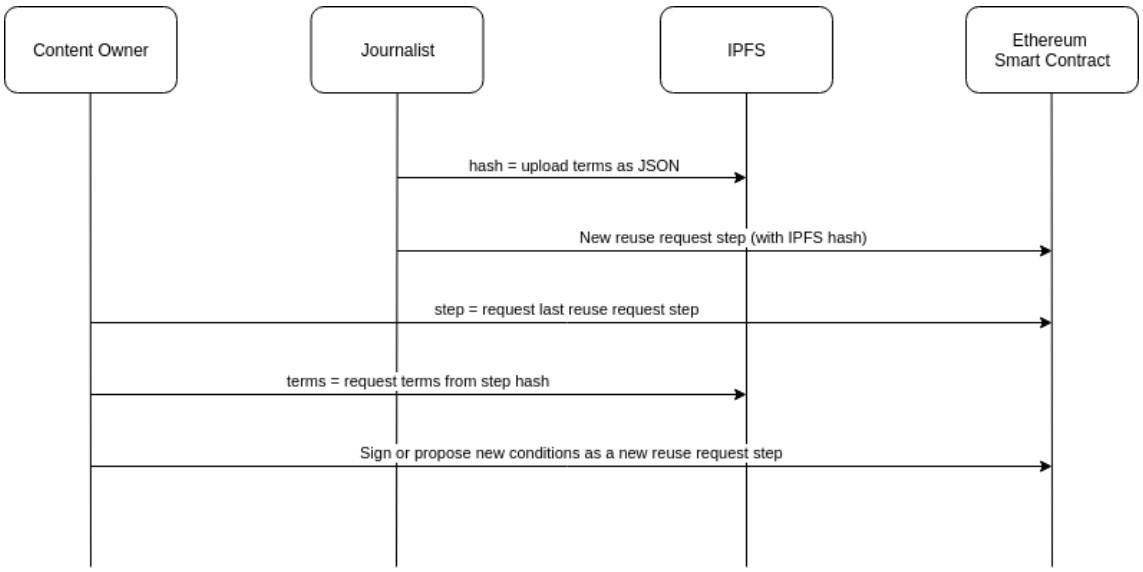


Figure 2.1: InVID Rights Management reuse request step flow

For the verification of identities, we will be using uPort and its developers API. uPort is a platform that facilitates identity management based on Decentralized Identifiers (DID). It is based on Ethereum and thus these identities are associated with one more Ethereum addresses, which can be used to send/receive funds like a

[2]Infura: `https://infura.io`

cryptocurrency wallet, but also to sign data, interact with smart contracts or receive attestations.

A uPort attestation is a JSON Web Token (JWT) with claims. A JWT is a self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed using HMAC, or a public/private key pair using RSA or ECDSA[12] (uPort uses this last one).

The JWT, signed by the sender/attester, can be stored and used by the receiver to prove a fact. The point of attestations is to create a social consensus where trusted actors attest things being true[13]. For example, a government could attest your real name or birthday, and the attestation alone would be enough to proof your official identity because it is digitally signed by the government.

A similar well-known project is MetaMask[3], a web browser extension that makes it possible to create your own vault of Ethereum addresses in a seamless way. However, it lacks DID and attestation management.

Decentralized Identifiers constitute a novel approach to *self-sovereign* digital identities. DIDs are fully under the control of the DID subject and are independent of any centralized registry. They are similar to Universally Unique Identifier (UUID), but DIDs are also URLs that relate a DID subject to means for trustable interactions with that subject, and they resolve to documents that describe how to use that specific DID[14].

The cryptographic complexity behind the DID makes it possible to use an identifier to link to data. Moreover, uPort generation of Ethereum addresses is based on the DID. Out of the box, the uPort API supports the disclosure of credentials, creating/signing transactions, and generating claims/attestations.

For the uSocial application, we will be needing the disclosure of credentials to display the current identity of the user (DID address) and uSocial attestations through the web client. Displaying uSocial attestations will make it easier for users to figure out what they have attested since uSocial attestations will have a specific format that may not be properly displayed on the uPort mobile application. Later, we will also be sending the current attestations to the attester server to concatenate attested values in the same signed JWT attestation, as it is more convenient to have the same entity attested values grouped together.

When it comes to InVID Rights Management, we will be needing disclosure of credentials with their attestations to authenticate guests to the platform. InVID

[3]MetaMask: `https://metamask.io`

will also be using uPort and the Web3.js library for the reuse request transactions flow.

Web3.js is the Ethereum JavaScript API which implements the Generic JSON RPC spec[15]. With Web3.js you can interact with the Ethereum network (send funds, deploy and call smart contract functions, listen to smart contract events, etc.).

While you can use Web3.js alone, which you often do when you work with extensions such as MetaMask, uPort libraries already use Web3.js under the hood. The common uPort functionalities do not require working with the Web3.js instance directly, and it is often just needed to carry certain operations such as interacting with smart contracts.

To write, debug and deploy the smart contract we will be using the Truffle framework along with Ganache.

The Truffle framework[4] can save you a lot of time when writing smart contracts; it automates compilation, testing, deployment, and migration of smart contracts. Alternatively, we could be using the Remix IDE[5]. However, while Remix makes it simpler to run, test and deploy simple smart contracts, it doesn't provide the right tools to work with a version control system nor it has migration built-in.

Migration is the process of *upgrading* a smart contract. While the term is exactly the same as the one we use for databases, the process is very different. Ethereum smart contracts are immutable, and there exists no versioning. Smart contract owners are meant to upload a new version of it every time they want to publish changes, and their users should then use the new version. Truffle provides a Migration.sol[6] smart contract that is responsible for keeping track of the deployed version so Truffle will only apply the rest of the upgrades, preserving from unnecessary gas cost.

For this reason, smart contract developers have to test their smart contract carefully and try to anticipate future modifications so that they can write the contract in a way that upgrading without data loss is possible and it is not too cumbersome. Other than the Migration version tracking, a common design pattern to support upgradability is by proxying smart contracts[16]. The proxy contract delegates calls to the proxied contract, as shown in figure 2.2, and the proxied contract can be upgrade by the owner without data loss.

Ganache is a software that mimics the behavior of an Ethereum network for testing purposes. Ganache, which is available as both a GUI and a command-line tool,

[4]Truffle: `https://truffleframework.com/truffle`

[5]Remix: `https://remix.ethereum.org`

[6]Migration.sol: `https://github.com/ConsenSys/truffle-webpack-demo/blob/master/contracts/Migrations.sol`

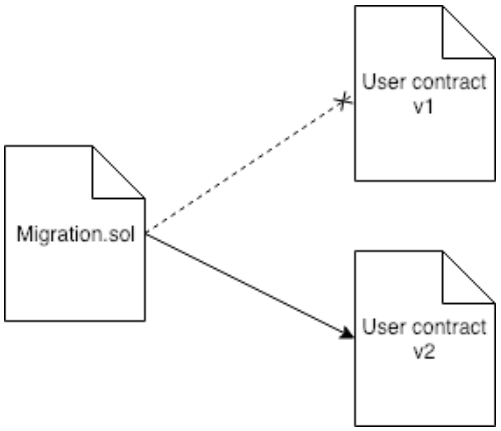


Figure 2.2: How proxying smart contract works

makes it possible to create various Ethereum nodes and miners, and it displays the transactions as they are being run. Ganache is often used with Truffle because it makes it possible to test smart contracts on a predictable way on a test network, rather than a live network on which you might get mixed results. Although we have been referring to the official Ethereum public network (Mainnet), there are many more Ethereum networks available. The most common ones are known as test networks which developers can use try out their smart contracts without incurring real costs, such as Ropsten or Rinkeby, which we will be used during the development of the project.

2.2 Web Infrastructure

We believe that the most adequate way to make our identity verification tooling is through a dedicated website. A website makes it straightforward for uPort users to access without the need of installing any specific tools or go through any cumbersome process. In fact, uPort users who access the web through the same mobile device as the one in which their uPort application is installed, will not even have to scan QR codes to pass information since the browser can reach the application directly.

Considering we are working with decentralized technology, the client will do a lot of the work, especially when it comes to fetching and displaying the data that resides on the uPort mobile application or on the Ethereum network. The server-side solution will be needed to verify the social networks identity and email and submitting signed attestations.

For the uSocial client side, we will be using React[7], a powerful Single Page Application (SPA) library that makes it easy to create custom interactive User Interfaces

[7]React `https://reactjs.org`

(UI).

By using React, uPort and Web3.js libraries, the client will display users' uPort information as well as their attested values. This part will not require server-side whatsoever which makes it more trustworthy since the users can inspect the exact code that is being run.

As we noted above, uSocial will also be needing a server-side to act as a point of trust. To do so, we will be implementing a Node.js API with Express. Node.js is a good option for building a robust, performant and quick API, but for this project in particular it is is the best option given that uPort only supports Node.js for the time being. We may also need Web3 and Web3 Utils. Web3 Utils makes it easier to deploy and read/write data from smart contracts, providing utilities such as `hexToAscii`.

On the other hand, InVID Rights Management will mostly need client-side work. The platform already supports creating and agreeing to reuse requests, both server-side and client-side, so our intent is to extend it to support Ethereum and IPFS as a secondary more trustworthy source.

The client-side should be able to make the process smooth and make it possible for the user to visualize what's on the Ethereum network and IPFS at any time. For example, the content owner should be able to read over the reuse request data stored in IPFS before submitting their approval to the Ethereum smart contract.

InVID will still need a server-side solution for the authentication system. Although uPort data can be accessed by the client through a disclosure request, InVID requires to match uPort identities with existing users to ensure that the reuse requests stored in Ethereum are being signed by the rightful owners. Part of the authentication procedure will consist in verifying the attestations previously granted by uSocial, specially e-mails.

2.3 Continuous Deployment

In the Ethereum section, we explained how Truffle simplified the deployment of smart contracts with their command line tools.

Continuous Deployment (CD) refers to the automatic deployment of source code onto production servers. This is interesting for various reasons:

- Eliminates the repetitive human intervention that the process of moving the source code to the production server(s) means.

- Having each feature, bugfix and minor enhancement deployed as source code is committed makes A/B testing[8] more reliable as it is easier to track the characteristics that make the most impact in the site User Experience (UX).
- Prevents deployment mistakes as only the Continuous Integration server will be authorized to access and make such changes on the production servers. Such automation makes it possible for developers not to store production keys, which also makes it safer.

The uSocial application will be Continuous Deployed in at least 2 different servers. The first will be for development purposes, that will work to verify that the features really work as intended. The second will be the production server, on which we are pretty sure that the features landing there work as intended. The development server should never be a replacement for testing though.

The InVID Rights Management source code is already Continuously Deployed, and we will not get into details about its current configuration. When it comes to deployment of the smart contract, we do not think CD is worth the setup investment. The smart contract should rarely change, especially given the complexity that upgrades can add to the original source code. There also does not exist the concept of environment variables and Truffle tools make it simple enough to carry the deployment locally. When we are deploying a smart contract or sending/receiving funds we are sending a transaction to a network node that will put it in the pool of transactions[17]. Miners will decide whether to include it in the next block, depending on the reward[18].

[8] A/B testing is an experiment where part of the visitors are presented a variant of the site (A) and others are presented a different variant of the same site (B). The one that gives the most conversion rate, that is calculated through a variety of metrics, is the best.

Chapter 3

Development

This chapter will detail the development process, from a mere idea to a functional system. A considerable part of the development process will move around uSocial, which we will build from the ground up, but we will also add the blockchain-related parts of the InVID Rights Management platform as we believe they are very important when it comes to the development of uSocial.

Doing so will help the readers understand the value behind the uSocial application and it will make it easier for us to support the different design decision that we will make. Moreover, we will be using InVID Rights Platform to showcase the integration procedure of uSocial verified identities.

We have divided this chapter into categories, that follow the chronological order of execution.

3.1 Methodology

We will be using Scrum, an Agile framework to carry out the development of the project in an incremental manner.

Scrum supports iterative and incremental upgrades. Its objective is to have the working version of the product in a short amount of time, making the product more and more complete over the time.

Furthermore, Scrum takes into account that requirements can change at any point of the project and we should be able to adapt to those changes just fine.

Scrum works with *sprints*. Each sprint is a set of time, between 1-4 weeks, in which a set of prioritized work is defined and has to be finished by the end of that time.

Any requirements change, or any work that was not completed during that time is reprioritized and set to be worked on on the next sprint. Our sprints or iterations are going to have a duration of 2 weeks.

Scrum will make it easy for us to visualize the project's progress, and we can progressively add new features that make our project more feature-rich as the basic functionality gets deployed onto the production servers.

The scrum framework also features the scrum master role as the player that coordinates the team's work and analyzes the work pace as well as makes communication swifter between the different roles involved in the project. We can consider the thesis directors as our scrum masters.

3.2 Temporal Planification

The time planification is a way to visualize the work that has to be done over the time.

Although we do not have a complete list of requirements, we are working with Scrum, so we can always add requirements at a later point in time. What we need prior getting started with the project is a list of meaningful requirements, so that we can properly prioritize the work that has to be done and we can preferably build a functional project within the first 2 iterations.

Moreover, our planification will have to take into account an estimation of the time required to learn the diverse concepts and frameworks that are new to us as well the documentation that we will be writing parallelly.

We define our initial product backlog, split by platform for clarity purposes, as follows:

uSocial

Id	Story	Estimation	Priority
1	As a guest, I want to authenticate into the platform.	5	1
2	As a user, I want to view my attested data.	5	10
3	As a user, I want to verify my email address(es).	10	10
4	As a user, I want to verify/connect my Google account(s).	10	10
5	As a user, I want to verify/connect my Facebook account(s).	10	10
6	As a user, I want to verify/connect my Twitter account(s).	10	10
7	As a developer, I want tools/API to help integrating uSocial attestations with my web app.	5	9
8	As a developer, I want uSocial to work on the mainnet, as well as test networks such as Ropsten and Rinkeby.	5	5

InVID Rights Management

Id	Story	Estimation	Priority
1	As a journalist, I want to authenticate into the platform with uPort.	5	1
2	As a content owner, I want to authenticate into the platform with uPort.	5	1
3	As a developer, I want the smart contract to handle the start and end of the negotiation, as well as the accepted rights.	6	10
4	As a developer, I want the smart contract to track each of the negotiation steps.	6	10
5	As a developer, I want IPFS the store a full copy of the requested rights.	6	10
6	As a developer, I want to provide journalists the option to carry the reuse rights request over blockchain by using uPort, Ethereum and smart contracts.	3	10
7	As a journalist/InVID user who is using uPort for rights request, I want to create an Ethereum transaction every time I accept/negotiate rights, as well as store it in the InVID Rights Management platform server.	10	10

The previous product backlog lists only features. A complete user stories list would also include[19]:

- Web operations: source code version control, developer's documentation, testing
- Deployment: CI, CD, orchestration, environment variables, domain
- Monitoring: logging, reporting tools
- Security: logging, access control, backup plans, disaster recovery
- Design: prototypes, network infrastructure
- Legal: Terms of Service, Privacy Policy, GDPR
- Miscellaneous: helpdesk, FaQ

While we will eventually have to work on some of the previously listed points, especially when it comes to web operation and deployment, we will focus on having the features ready and accessible. The uSocial platform's goal for this thesis is to be a working prototype which is able to demonstrate how the verification of social networks for decentralized networks could work, rather than a production-ready or commercial application.

Either way, in the iteration sections we will be documenting the most important points when it comes to anything related to the making of the uSocial platform and integration with InVID Rights Management.

3.3 Setting up uSocial

The first development cycle started by setting up the development environment. A good project setup can save us a considerable amount of time when developing the project as well as make it easier for anyone who wants to run our project or/and contribute to it.

We also wanted to set up CI and CD as early as possible.

CI provides feedback about the state of the source code in real time, which makes it easier to spot the parts that are not working as intended. Otherwise, we would have to figure this out by using version control tools such as `git bisect`.

Then, CD makes it easier for us to visualize the results in a production-like environment as the platform is being developed, as well as enabling us to share these results with the stakeholders.

Version Control

We are using Git and GitHub as our version control software and source code hosting respectively.

We are working with two branches, development and master, based in *A Successful Git Branching Model*[20].

The development branch always has the latest features, while the master branch is updated less often in exchange for more stability. We use the master branch deployments to showcase the project.

Monorepo

One of the first decisions that we made was uSocial to be a monorepo: the client and API to reside in the same repository. However, that does not mean that the client and API are tied together. In fact, they remain as independent projects that can be executed, built and tested independently; but this makes it more natural for versioning, changes, and deployment of new features as we can simultaneously push new changes that affect both the client and API at the same time.

Working with a monorepo also makes it easier to run both a the same time, run E2E tests and share source code between them. Remember that both the API and the client understand JavaScript.

For JavaScript repositories, Lerna[1] is one of the most popular choices to manage multiple packages. It makes it possible to view, install, run and deploy all packages at once. We make use of Lerna to handle the installation of packets, booting applications and serving production-like versions. The deployment part is handled by Travis which we will get into details afterward.

Client and API

To get React bootstrapped without having to deal with WebPack configuration, we are using *Create React App*. Create React App supplies a basic project structure with Hot Reloading, testing and production build.

React Hot Reloading refreshes the website as you save changes as well as it preserves the state.

We are working with React 16.7 alpha which gives us access to the upcoming React Hooks.

For the API, Node.js basics are easy to set up since it doesn't require transpiling the source code like React. As of 2019, client-side applications should be transpiled

[1]Lerna: `https://lernajs.io`

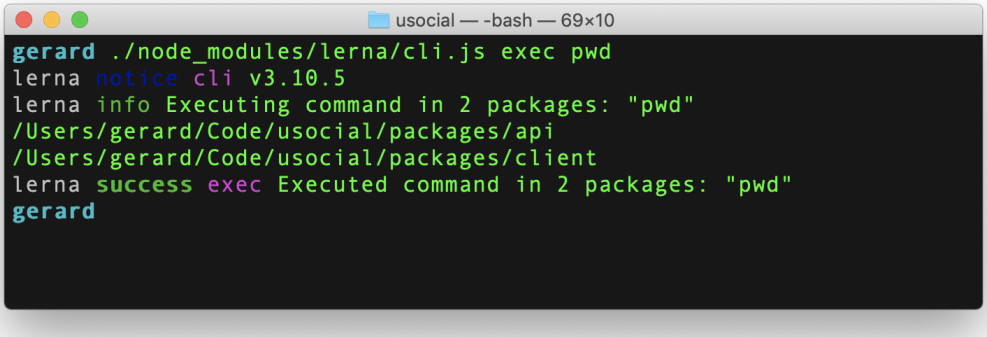


Figure 3.1: Lerna enables executing a command to multiple packages at once

to either ES5[2] or ES6 to widen browser support.

Most of the Node.js setup will be done as we need it, based in Express Starter Kit[3] folder structure and configuration. To start, we will be configuring an Express server that will act as the communication point between the client and the API.

We are working with Node.js 11 while giving support to 10 (the Long Term Version).

We also enabled the support of environment variables through .env files. Dotenv files make it easier to set environment variables that persist over sessions and are project specific. While the client will have very few variables, such as the URL of the API or the port that it has to listen to, the API will have a few, some of which are intended to remain private: host addresses, port, email host, username and password, uPort private key and social networks client ID and secret keys.

Our final folder structure looks like the following:

```
.
|-- LICENSE
|-- Procfile
|-- README.md
|-- lerna.json
|-- node_modules
|-- package-lock.json
|-- package.json
`-- packages
    |-- api
```

[2]ECMAScript was created to standardize JavaScript. ES5 or ECMAScript 5 is the JavaScript specification released in 2009.

[3]Express Starter Kit: `https://github.com/zurfyx/express-api-starter-kit`

```
11      |   |-- index.js
12      |   |-- node_modules
13      |   |-- package-lock.json
14      |   `-- package.json
15      `-- client
16          |-- node_modules
17          |-- package-lock.json
18          |-- package.json
19          |-- public
20          |   `-- index.html
21          `-- src
22              `-- index.js
```

CI and CD

One of the most popular Continuous Integration services for GitHub is Travis, which provides unlimited free runs for public repositories.

We are using Travis to ensure that our code works as expected, ESLint (the JavaScript linter) passes and production versions can be built just fine for each of the commits. Travis runs automatically when a new commit is pushed onto the GitHub repository.

Travis also handles deployment for us. When a commit build and testing phases are successful (builds without errors and linter and tests pass), Travis will deploy it to the hosting provider, the responsible for serving the project on the Internet.

We have chosen Heroku to be that hosting provider, even though we might eventually switch it to a dedicated server such as the one on which the InVID Rights Management is currently running.

Heroku offers a free limited monthly runtime for our projects, which should suffice for now. Deployment with Heroku is a slightly different from deploying onto a VPS (Virtual Private Server) or dedicated server in that you have to use their command line utilities and specifications. For example, you have to write a Procfile configuration file to run a specific command after waking up from sleep or executing for the first time after deployment.

Our configuration requires:

- Heroku environment variables that we can set up through their cli or GUI.
- package.json `heroku-postbuilt` script to build the production version, that will have to be executed once every deploy. Note that the production build takes into account environment variables that only the Heroku instance is aware of.

- Procfile script that determines how to program is executed.

For the API, the Procfile simply points Node to execute the main file. The client requires an HTTP server to be able to run the production files since Heroku provides no static file hosting, unlike GitHub pages. Package *serve* is an HTTP server oriented towards SPA sites, which not only hosts files but also forwards not matching URLs to `/index.html`.

A Heroku limitation is that we can only open a port per application. While our API could serve the client, we would rather they are not tied together. Ideally, we would have a software managing that for us, but we kept it simple and published both separately. To undergo that limitation, we simply created two application per branch (one for the API and another for the client).

That makes four Heroku applications:

- uport-social (`https://uport-social.herokuapp.com`)
- uport-social-api (`https://uport-social-api.herokuapp.com`)
- uport-social-dev (`https://uport-social-dev.herokuapp.com`)
- uport-social-api-dev (`https://uport-social-api-dev.herokuapp.com`)

With the basic setup ready, we were ready to work on the uSocial platform features.

3.4 Authentication

The first user story that we undertook is "As a guest, I want to authenticate into the platform". Not because it has the most priority but because it is a dependency for most other features.

Authentication on uSocial relies solely on uPort, and uPort already provides libraries to do so on the web: uPort Connect and uPort Transports.

uPort Connect is the uPort Swiss Army Knife for front-end and makes use of uPort Transports for communication operations. It handles:

- Request disclosure: retrieve user data stored in the mobile device
- Sending JWT messages to a server
- Request signatures
- Read smart contracts and create transactions to interact with them

uPort Connect can provide the full-fledged Web3 instance that it uses to handle some operations behind the scenes, but we will get into details about that when we need it.

Let's go back to the uPort authentication. As you may have noticed on the uPort Connect features list, there is no authentication. The closest one to authenticating from the list is request disclosure.

And that is because uPort does not offer the traditional server-side authentication; there is no session and every time we want to retrieve new or updated data we the user will have to approve it on their mobile application. At first, it just seems to add a redundant layer on top, given the poor synchronization options. However, there are advantages for which it makes sense to keep it.

Next, we will cover some of them.

UX

Most websites will require your account credentials before enabling you to take any actions over your account details or create content on their platform.

For this reason, we can say we are used to authenticating, and so we believe we should treat our uPort disclosure credentials as a way to authenticating into the platform since it enables further restricted actions.

List current verifications as well as displaying basic profile data

A request disclosure makes it possible for us to request most account information stored in the user's mobile device. Exceptions include wallet private keys and settings.

With the disclosure request information we are able to build the user's profile page and make them feel like they are signed in.

Displaying users' profile does not really add much value since they can already check a complete version of them on their mobile devices. But the attestations do because they each provider (such as uSocial) has got their specific format, and we can make their visualization a lot more user-friendly.

With a single disclosure request disclosure, we can request both the profile information and attestations in the same petition.

Displaying user attestations, as shown in the user stories table, is one of the top priorities of the uSocial site.

Server-side attestations will require a DID

In order for us to grant verifications to certain uPort account, the user has to have provided the DID first. Entering the DID by hand is an option, but not very practical and error-prone considering they are long and consisting of random alphanumeric characters.

The authentication request disclosure already grants us the DID which we can later transmit to the API to generate the attestations.

The uPort request disclosure we ended up with consists of name, avatar, email and uSocial verifications (under the name *Usocial Identity*) as well as the default uPort public information that comes with every single request, such as the DID.

```
1 const disclosure = {
2   requested: ['name', 'avatar', 'email'],
3   networkId: 'rinkeby',
4   notifications: true,
5   verified: ['usocialIdentity'],
6 };
```

Listing 3.1: uPort client request disclosure

A couple notes on the snippet above. First of all, we are, by default, operating on the Rinkeby test network. Test network transactions are free of charge, and we are initially aiming at developers. Secondly, enabling notifications can come in handy. By default, uPort mobile app requires scanning QR codes containing a JWT for every single operation. Requesting notifications grants us a token that makes it possible to send requests on the mobile app at any point; they come as push notifications which the uPort mobile application immediately displays enabling us to do the same operations with fewer users' effort.

This feature is fully client side, and other than uPort specifics, we had to set up routing, build an authentication screen and a post-authentication dashboard and a mechanism for controlling signed in users data.

We could argue that the last item is optional since uPort Connect already stores a copy of the requested disclosure into local storage, under the name `connectState`. However, we have no guarantee their copy will always behave like that or that it is compatible with modifications we can make to suit our needs in the future.

We decided to integrate the login into the homepage, shown in figures 3.2 and 3.3, instead of building a dedicated sign in screen as it makes it easier for newcomers to spot it and at the same time it helps them figure out what the site is about, even if we have not added much information about what uSocial is about yet.

When it comes to the dashboard design, we wanted something easy to use to support

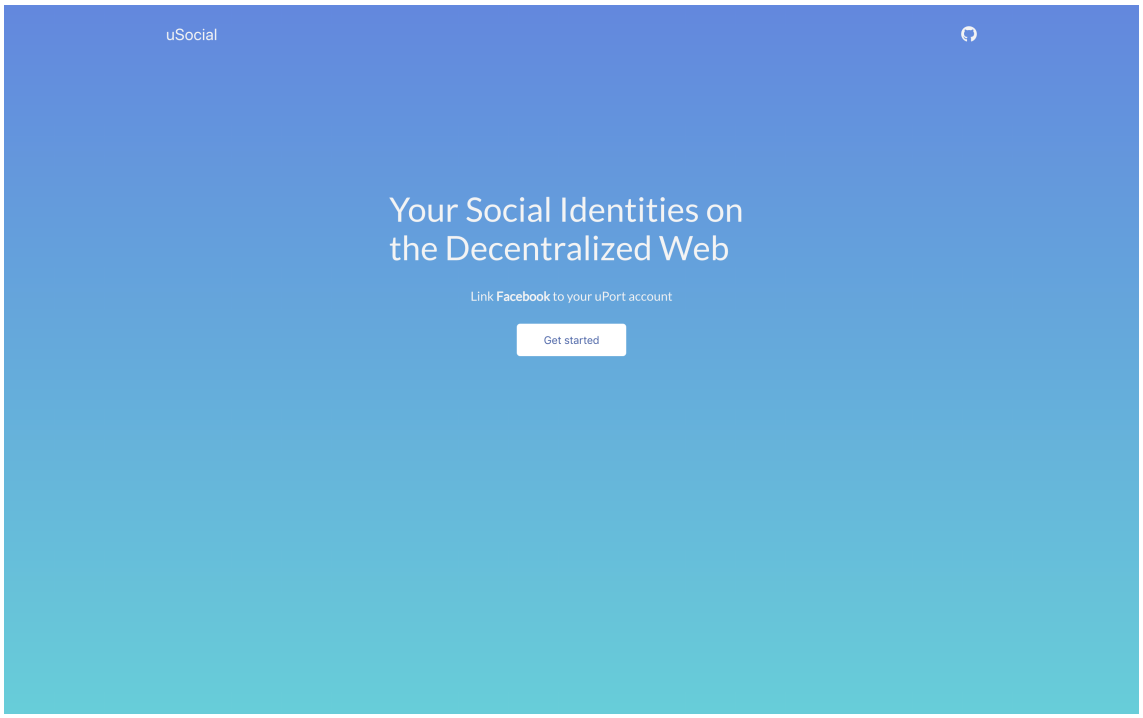


Figure 3.2: uSocial homepage

at least the following actions:

- View attestations
- Add a new connection
- Integration with third party websites
- View profile
- Sign out/clear all user data
- Change network

A common UI choice, used by Facebook Developers and Digital Ocean, is to have a side navigation menu displaying the different site tools, and a top navigation bar displaying the signed in user and complementary navigation options. And that is what we did, shown in figure 3.4.

React makes building complex interfaces very convenient through a tree-like hierarchy of components, such as figure 3.5, and naturally being able to split components into smaller ones. A dashboard can simply consist of a set of components, each of them may have smaller components recursively.

Similarly, React Router provides wrapper components that will conditionally render your routes. React Router also offers High Order Components to read/change the

Figure 3.3: uSocial homepage displaying a uPort JWT code with QR

state of the browser history.

A High Order Component is a function that takes a component per parameters and returns a new component. The term comes from the Computer Science term High Order function which is a function that takes functions per parameter and returns a function.

When we are talking about routing, we have to take into consideration that SPA (Single Page Application) sites only have one endpoint (index.html). An HTTP server such as *serve* forwards 404 requests to it and it is React Router which eventually, by using location or browser history, displays the appropriate component. The browser history API also makes it feasible to dynamically change the browser displayed path without reloading the page.

To handle the authentication state we are using React Context. React Context is simpler to set up than alternatives such as Redux or MobX and it pretty much suits our needs: a global setter and getter of the user data object. The fact we cannot make modifications to the user through the client makes it exactly a setter and getter, manageable through React Hooks.

```
1 const [user, setUser] = useState({});
```

Listing 3.2: React Hooks to set/get user state

The feature that we are missing is the ability to back that state on persistent storage,

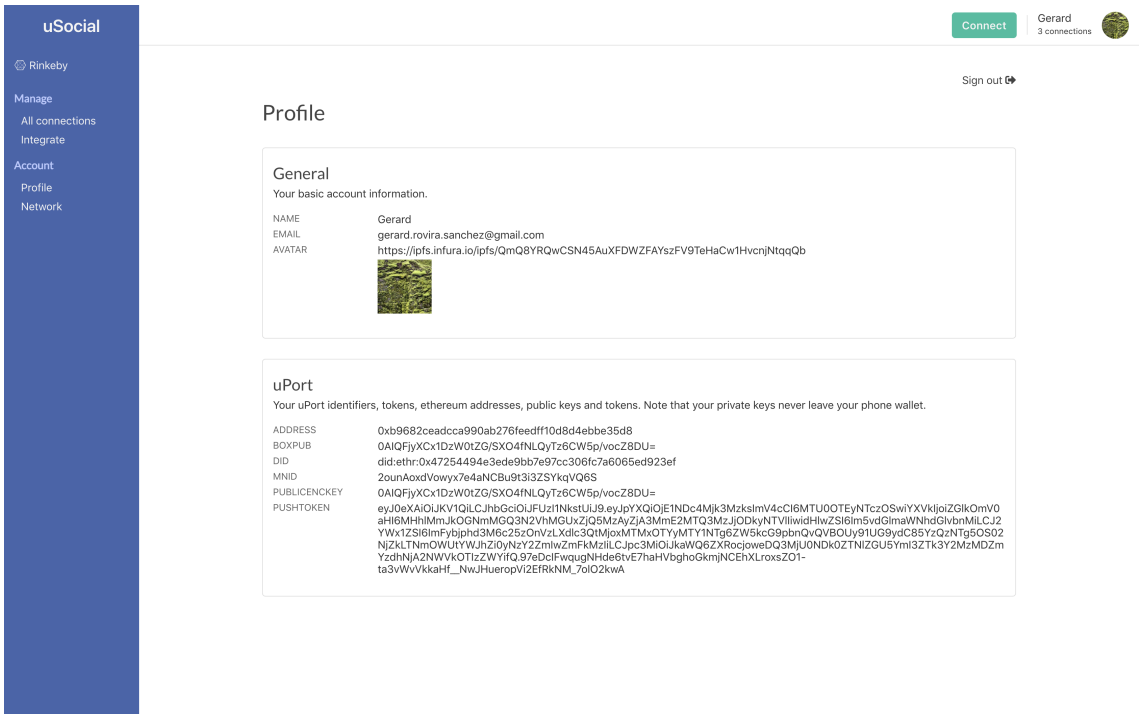


Figure 3.4: uSocial displaying profile inside dashboard

such as local storage. But we can implement a proxy on top of hooks `useState` for that like the following:

```
function useLocalStorage(key, initialState) {
  function localStorageSet(state) {
    const json = JSON.stringify(state);
    window.localStorage.setItem(key, json);
  }
  function localStorageGet(key) {
    const json = window.localStorage.getItem(key);
    return JSON.parse(json);
  }

  let state = localStorageGet(key);
  if (state === null) {
    localStorageSet(initialState);
    state = initialState;
  }
  const [item, setItem] = useState(state);

  function proxySetItem(newState) {
    localStorageSet(newState);
    return setItem(newState);
  }
  return [item, proxySetItem];
```

```
23 }
```

Listing 3.3: `useLocalStorage` implementation

Replacing the above call with the following:

```
1 const [user, setUser] = useLocalStorage('user', {});
```

Listing 3.4: React Hooks to set/get user state while backing it up into local storage

We have to note that for a secure successful signout we have to delete both our local storage entry as well as uPort's one, automatically created through the uPort Connect library.

That concludes the authentication feature for the uSocial site. Next, we will be working on connections.

Figure 3.5: React hierarchy for profile page profiled with Chrome devtools and React Developer Tools

3.5 Connections

Connections are the core of the uSocial platform. The point of connections is to verify their identities on social networks and emails. We often refer to them as verifications or attestations (even though that attestations can have multiple attested connections/verifications in the same JWT), and that is because we are in fact verifying the identity behind social network accounts and attesting such fact via signed tokens. We name them connections because we are in a way simulating what centralized applications do when they connect various third-party platform accounts to one to enable performing authentication to their site through multiple sites.

Third party platforms have to have access to such verifications to perform restricted actions, such as associating various credentials to the same account.

For InVID, content owners can authenticate to the platform using Facebook, Google, Twitter, and YouTube. These social networks are used to verify that they are the publishers of the videos the journalists want to request rights over. So as to have a unique identity on the site, and not a different identity for each of the social networks, we are associating them to the same account according to their account's email (unless the authentication is performed when they are already signed in).

The addition of raw uPort would not only mean a new account on the system per uPort DID but also that we would not be able to associate the media ownership due to the lack of social networks information.

Having email verified by uSocial means that we are able to treat the uPort authentication just like the others. To make it possible for content owners to demonstrate that they are the rightful owners behind social media videos, they have to access with the social network account on which they were uploaded as well as uPort to enable handling the reuse request over Ethereum.

3.5.1 Generating attestations

As stated in the State of the Art section, uSocial acts as the point of trust and gives third-party platforms the certainty that the validated information is correct.

To do so, uSocial makes use of attestations, arbitrary pieces of data identified and signed by the DID owner. The fact that they are signed prevents anyone from tampering with them since only the DID owner could have have generated such attestations. Attestations, as JWT tokens, can also have an expiration date. Attestations are stored on user devices, and each receiver is responsible for keeping them

safe. This makes it possible for users to choose what and when they want to share such information.

Our attestations will solely contain the identifier of the account in each of the social networks. The identifier alone is enough for developers of external sites to be able to do the identity matching, and they can always obtain further information through the social network API if needed.

Just like InVID Rights Management, we are supporting Facebook, Twitter, Google, and email.

Throughout the implementation of the various social network logins, there is a key point that we kept present at all times. **The API must not store any user identifiable data**, not even under a temporal web session. It should solely serve as the point of trust that generates attestations and should be completely stateless.

Email

The first connection that we have gone through is email. We believe it is critical for uSocial users to be able to verify one or more email addresses.

The email verification can be done in two isolated steps. First, the user inputs their email on the uSocial site. The user is sent a QR on their email that will lead to the final API URL responsible for generating the attestation.

Figure 3.6 shows the complete email verification flow.

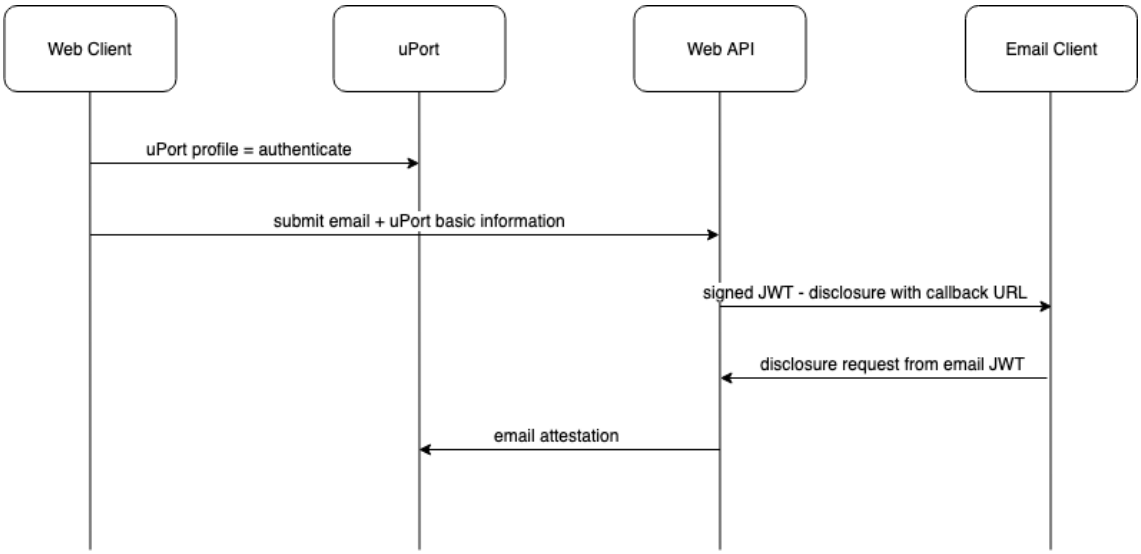


Figure 3.6: uSocial email flow

The signed JWT (stored inside a QR image such as figure 3.7) contains the email address that we are attesting, and the API will make use of it to generate the

attestation, after verifying that the signature is correct. That makes it stateless, and also possible to be completed on an external device, such as their mobile phone. Otherwise, the verification would have to be completed with the same computer that initialized the request.

Figure 3.7: uSocial mail sample with its JWT decoded content

We are storing the verification under *Usocial Identity* name, with a duration of 365 days. The end duration, just like with GPG, is recommended if you do not have a revocation certificate[21]. This way if the attestation is leaked it will eventually expire.

The attestation is sent to the uPort mobile device through push notifications, and it looks like figure 3.8. The token required to do so is requested on the same JWT, destroyed after the request is complete.

Facebook and Google

The second kind of social network connection is OAuth2, which includes Facebook and Google.

Just like with email, we want to make sure no personal information is ever stored in the API, but OAuth2 makes it pretty straightforward.

Passport[4] is a Node.js library that specializes in authentication. It has over 500 authentication strategies. While Passport was meant to do server to server authentication, on which it takes care of the whole flow, we will be using it to a certain

[4]Passport: `http://www.passportjs.org`

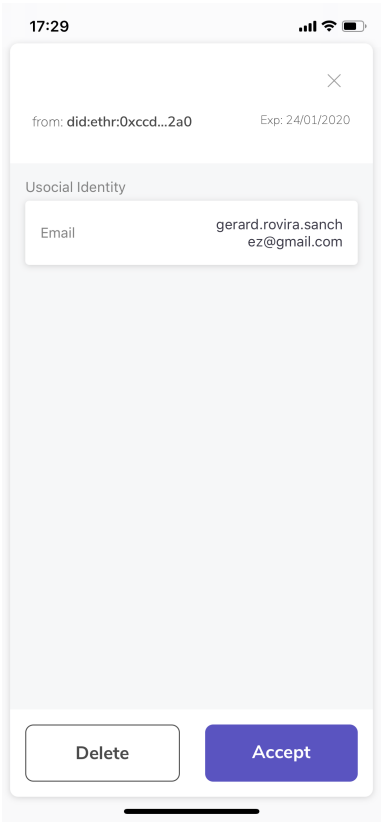


Figure 3.8: Email attestation through push notifications

extent for the OAuth2 flow, shown in figure 3.9. For OAuth2, the reason why we cannot fully use Passport is because the API requires uPort information to push the attestation, so we will have to move part of the flow to the client.

OAuth2 is simpler because the client can obtain the authorization grant token without a previous call to the API. It is afterward when the API will have to do two calls to the social network authentication server in order to retrieve the user profile information, but Passport can do that for us.

When Passport is done retrieving the profile information it will make use of basic uPort information, passed with the same API call, along with the social network profile ID to generate the attestation.

The uPort information consists of:

- DID: The receiver of the attestation (who has been verified to be the owner behind that account).
- PublicEncKey: uPort public encryption key.
- PushToken: PushToken makes it possible for us to send the attestation through Android/iOS push notification.

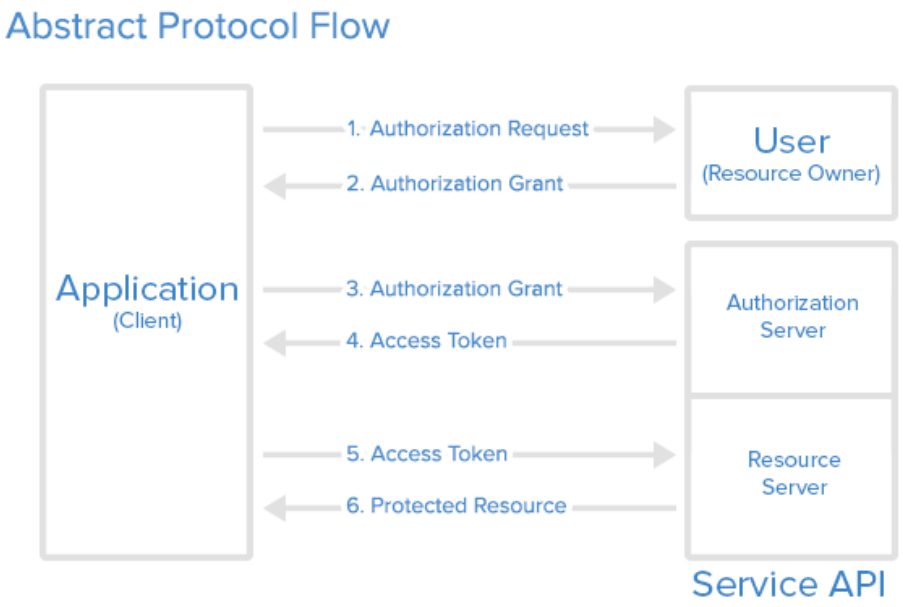


Figure 3.9: OAuth2 flow, from [1]

The final uSocial OAuth2 flow is shown in figure 3.10.

The OAuth2 state is a CSRF mechanism[22]. OAuth2 state is generated once the users click on the connect button, stored in the session storage, and is checked to be the same afterward.

Twitter

Twitter does have OAuth2, but it is for application-only authentication. User-based authentication requires using their OAuth1 authentication. OAuth1 flow is shown in figure 3.11.

OAuth1 is more complex than OAuth2 for us in that is it by design stateful. The client cannot receive the required token verifier without a previous API-acquired token. Additionally, the server has to keep itself a secret that has to be used afterward with the client token verifier in order to obtain the final tokens so as to be able to use the API.

Unlike with OAuth2, we cannot make use of Passport for OAuth1 because it requires a session to work. While their requirement makes complete sense because it is meant for server to server authentication and they have to keep data stored somehow, we were looking for an alternative which kept the server 100% stateless.

To do so we will be using NPM oauth[5], which makes it possible to customize the various steps of the OAuth flow so as to suit our needs. The oauth library will help us with the various requests as well as the generating the signed HMAC-SHA1

[5]oauth: `https://www.npmjs.com/package/oauth`

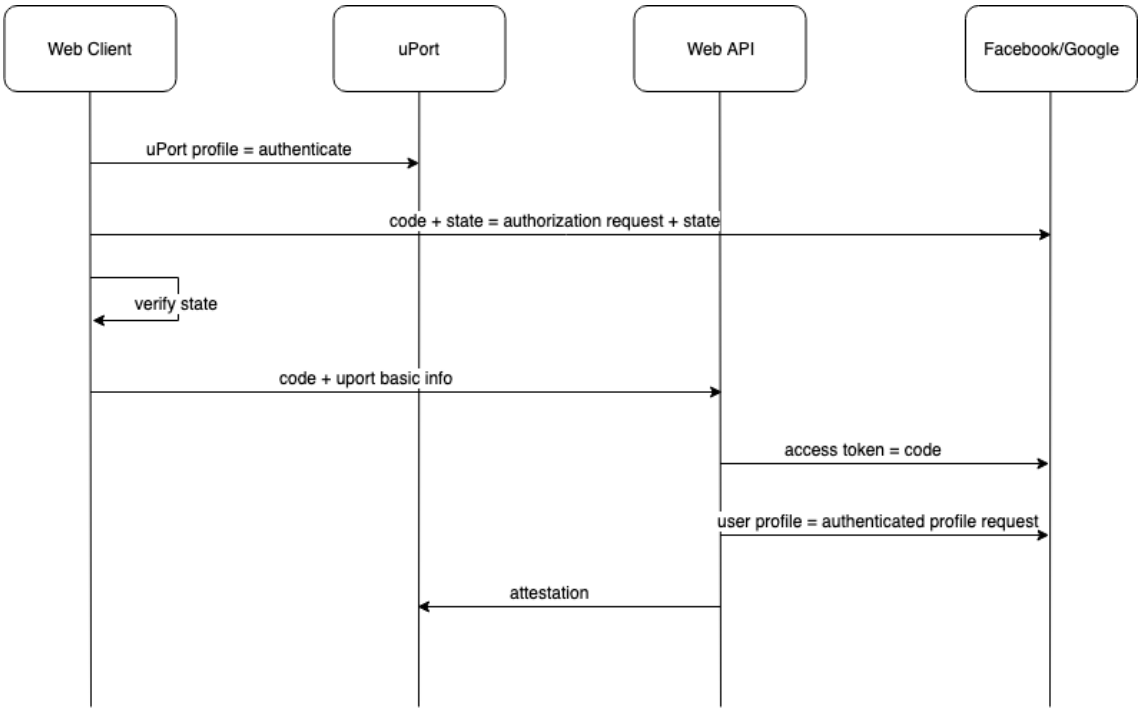


Figure 3.10: uSocial OAuth2 flow

required to obtain the initial token and secret[23].

To solve the secret key associated with each request, commonly stored as server session, we will be sending it to the client along with the needed token encrypted. The Node.js crypto library makes it possible to generate encrypted messages using AES256, that can later be decrypted using the same IV and key that were used to generate it.

On the client, data is kept on the session storage during the redirection to the social network site, just like OAuth2 state.

The full uSocial OAuth1 flow is shown in figure 3.12.

Just like with OAuth2, we send the basic uPort information along with the last API call so that the attestation can be generated and pushed to the user's uPort application on the mobile devices.

3.5.2 Keeping Usocial Identity attestations together

The most basic implementation of uSocial attestations consists in pushing a new attestation to the user's mobile device for every new platform verified.

While this one works, and a developer could still figure out the valid attestations

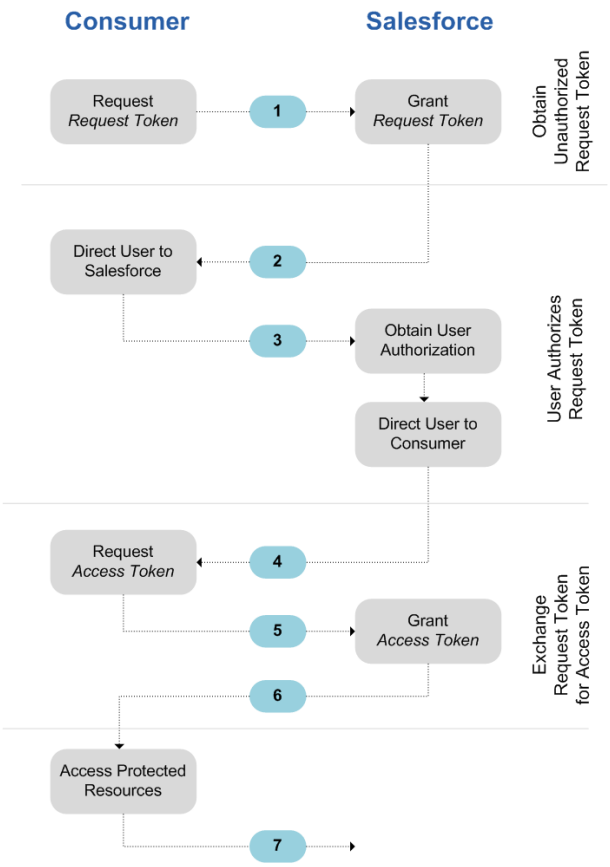


Figure 3.11: OAuth1 flow, from [2]

without extra complexity, it is not the most convenient for a user who has attested many different items using a single uPort device. It makes it difficult to figure out what they have attested and where from the uPort application.

Our current implementation takes the previous attestation, verifies it and creates a new attestation that includes previous attested values as well as the current one that is being verified.

Just like an external site would do, to prove the authenticity of the attestation, we are verifying its JWT signature. If everything is correct, we create and push the new attestation to the user. We will get into more details about it in the Integration section.

For OAuth authentication, the previous attested values are passed along with the last OAuth callback request. For email, it is sent along with the callback URL on the QR code sent to their email.

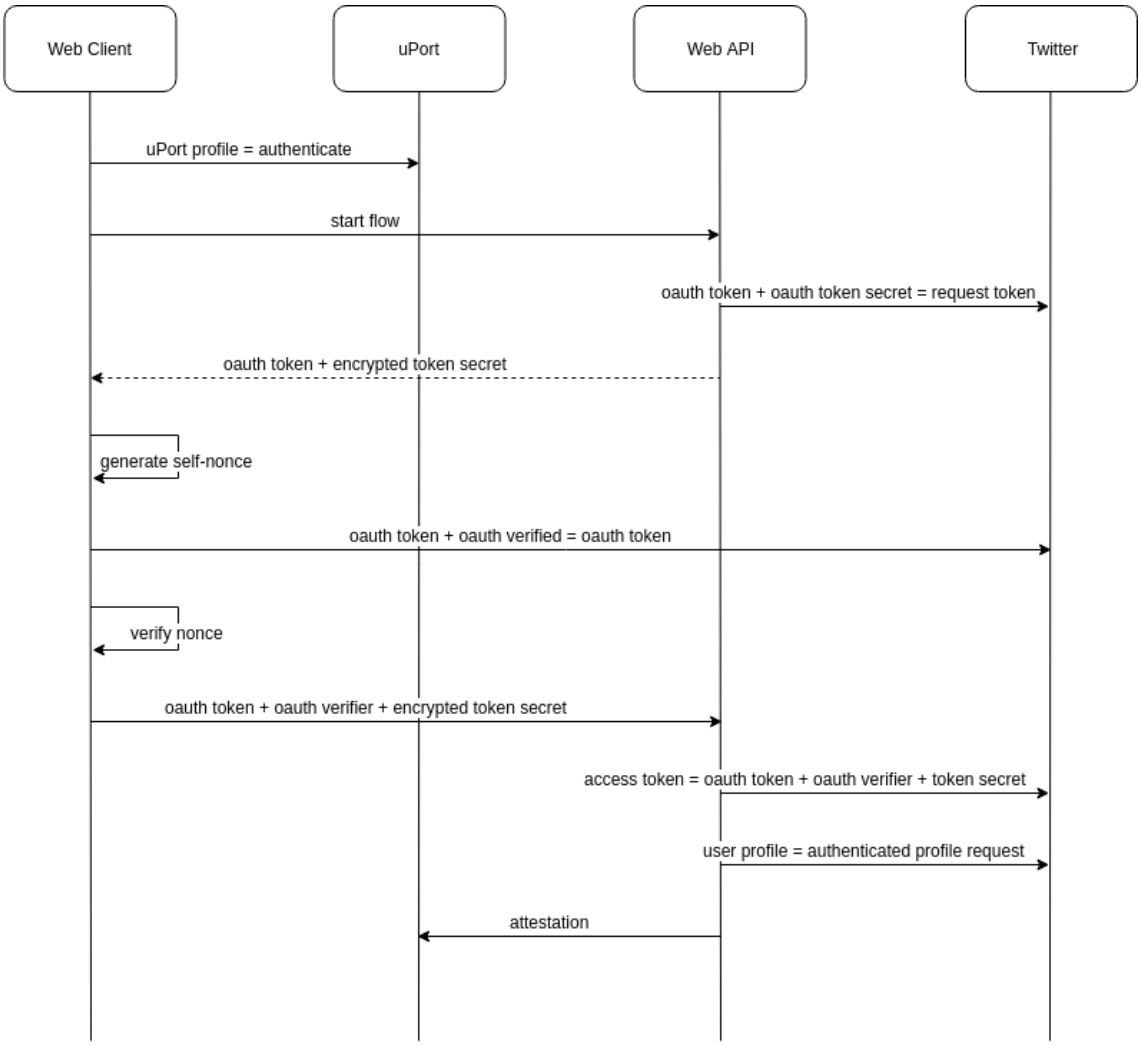


Figure 3.12: uSocial OAuth1 flow

3.5.3 Viewing attestations

The last step remaining is users to be able to see their uSocial attestations on the site.

The profile information we presented in the authentication section already carries attestations identified by *Usocial Identity* name. Printing resulting attestations is sufficient for the average user. But we went further than that and displayed valid attestations as well as attestations that had not been created by Usocial Identity. The initial implementation had the validation integrated with the same web client. But later, in the Integrate part of the project, we will see how we make this a lot simpler through a dedicated library that is also the library that we suggest third-party developers to use on their sites to validate uSocial attestations.

Generally speaking, this is a pretty straightforward part of the project. We are just

printing data. However, there is a minor issue in the displaying of connection, and it is due to the origin of the data. There is no way the retrieve new data behind the uPort account without a disclosure request, which requires the intervention of the user.

Most of the times, the data will already be up to date. But, there are few specific cases when the browser stored connection will differ from the ones stored in the mobile devices:

- The user has moved back to computer A, after attesting some data on computer B.
- The user has verified an email address, which has no connection with the web client.
- The user has removed an attestation.

A stateful server-side solution would have easily solved the first and second points, but it would have been at the cost of running a stateful server.

A second valid solution would have been to force a disclosure request everytime the user wanted to check their connections on the site, but this one comes at the cost of the convenience. To ensure consistency, we are already doing so prior to any connection.

Our current solution presents the users the option to refresh their connections through a new request disclosure button on their dashboard, a middle point between consistency and convenience.

3.6 InVID authentication

Having the attestation system ready (uSocial), we were ready to start with InVID Rights Management platform. To give an overview of the remaining work on that platform:

- Authentication with uPort
- Ethereum smart contracts
- Content Owners / Journalist rights negotiation backed to Ethereum and IPFS

The authentication, similarly to uSocial, starts on the client, through a QR code that users will have to scan in order to get authenticated to the site. However, this time the disclosure request JWT is not kept on the client. It is sent to a Node.js dedicated server (InVID Rights Blockchain) for blockchain purposes.

The Node.js server will verify the attestation the authenticity of the attestation generated by uSocial (if any) and return a new valid session key.

Having said that, you might wonder why we need server-side for a blockchain mechanism when ideally blockchain should be client-side only. There are two deeply related reasons why we have to do so:

A reuse request has a from and to addresses to determine the parties involved in the agreement. The content owner signing the reuse request has to be exactly the user behind the media the reuse request is about. For this reason, the journalist has to be somehow aware of the content owner address.

A safe mechanism to store both parties addresses is by using a server-side solution which we rely on. While JWT signatures make it possible for a client to verify signatures such as uPort disclosure requests or attestations, it is impossible for two clients that do not know each other to communicate directly between them. InVID server is both known and reachable for both of them. Alternatively, the reuse request negotiation could be done over smart contracts that operate on solely the reuse request ID without specifying the from/to of the parties involved. However, that is a bad idea, because anyone could easily argue that smart contracts data is public and anyone aware of the contract address can create transactions over its smart contracts methods.

We also require our own server-side because media ownership validation is executed by InVID. InVID stores the ownership behind the requested social media, which makes it feasible for us to match the identity of the social network account and uPort via a common verified attribute.

At this point, you could argue that by using our server-side solution on top of blockchain makes blockchain useless since we are establishing our own point of trust for the identity matching, but that is not necessarily true. The identity matching and storage of the various user account addresses enable a journalist to start reuse request over blockchain, and content owners to keep track of the contract metadata, but InVID still makes verification possible through client-side.

At each reuse negotiation step, the signing party (automatically) verifies that the other party identity and reuse request details uploaded on the blockchain are correct according to the InVID Rights Management state. The uploaded identity should at least consist of the address that is being used to negotiate, and an attestation that proofs their identity. While there is no way for users to do the media verification directly from the client, the media platform can cooperate in case there is ever a conflict. What is important to note here is that through the identity uploaded on

the blockchain, we can eventually trace back to the real owner behind the media content (see figure 3.13).

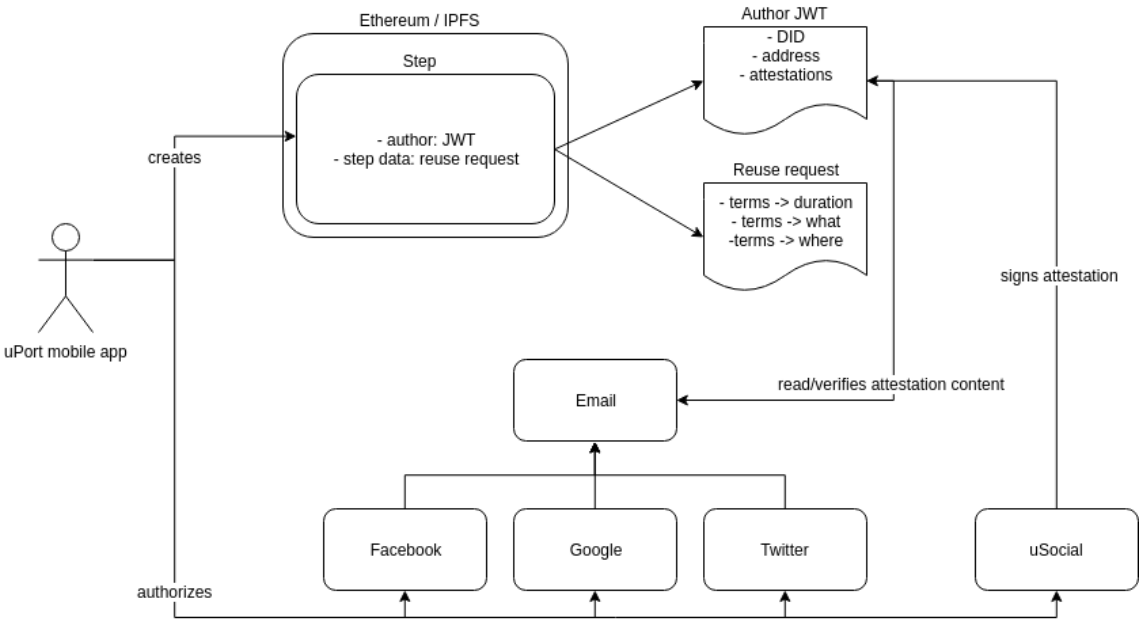


Figure 3.13: Schema of how the decentralized stored data pinpoints to the real user

We will get into more details about the necessary data to trace back to the real user when we cover the negotiation mechanism.

The InVID authentication flow starts from a disclosure request token (initiated on the client) to a session token, sent from the server back to the client. Contrary to uSocial, the disclosure request will have a callback URL pointing to the InVID Rights Blockchain API which the uPort application will call instead.

What makes it more complicated than the average login is the fact that the web client is not the one sending the request, the uPort application is. Hence, we have to map the uPort request to the web client that initiated it. And that is obviously a required step, otherwise, two users simultaneously authenticating would collide between them and receive misplaced messages, such as another user session token.

The full InVID authentication flow can be seen in figure 3.14. For the disclosure response JWT, note that it is the uPort mobile application after scanning the QR code the one that is sending the disclosure data to InVID Rights Blockchain.

The matching can be done with a temporal UUID (Universally unique identifier). Remember that prior the authentication there is no way we can match them through user identifiers, plus a user identifier would not make it possible for us to distinguish between various browser sessions. This UUID will be carried by both the mobile uPort application and the web browser to communicate between them.

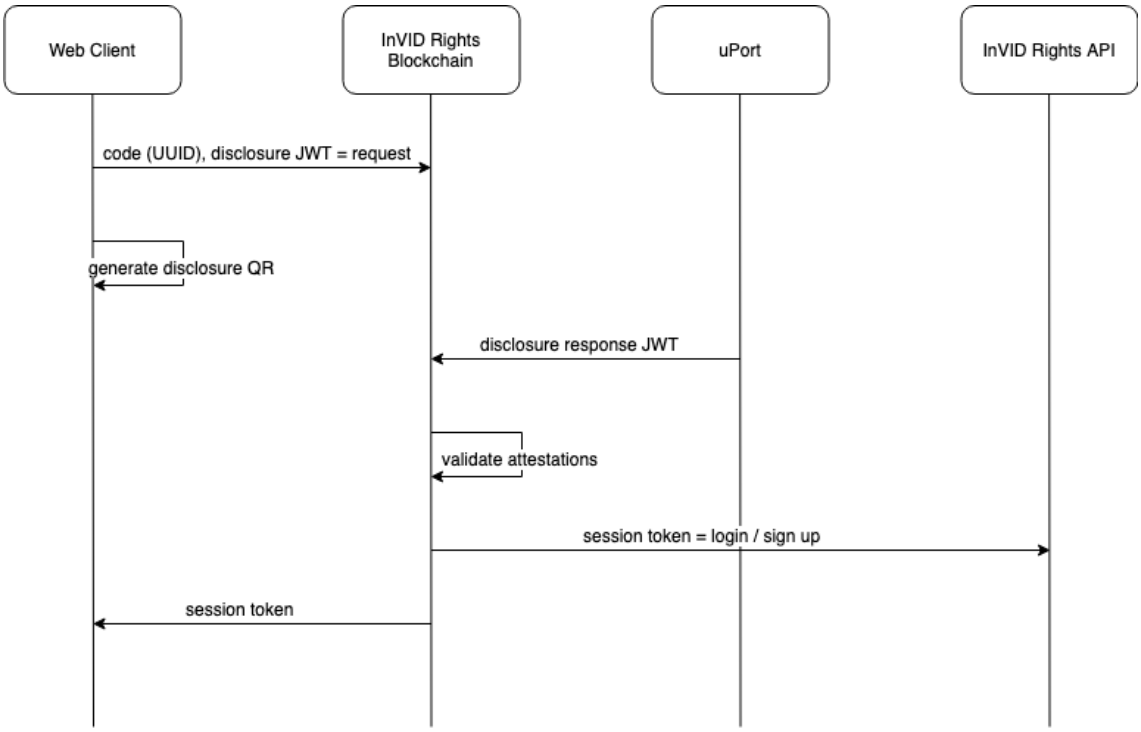


Figure 3.14: InVID authentication flow

The mobile UUID is passed along with the authenticate request disclosure on the QR code displayed on the site. I.e. `https://rights-blockchain.invid.udl.cat/contentOwner/disclosure/callback/d40d1334-93b9-40e7-bcc4-0dad865fa0d4`

The communication for the client to listen to login updates, triggered by the uPort mobile application, can be done in a couple ways. The first one is pulling. The client can repetitively query the server whether it has updates on the UUID code established until the authentication has completed.

There are two downsides with the pulling approach. First of all, pulling is more expensive and there is always a noticeable delay between requests. Secondly, a polling approach requires the server to keep a store of messages, that the user can pull from (similar to how POP for email works). It also requires the server to set up a cleanup system to destroy the messages after a certain timeout.

A WebSockets approach makes it simpler and smoother for us. The bidirectional connection makes it possible for the server to send messages to the web client as soon as a response is available. Hence, by using Node events we can keep the authentication stateless.

```
1 const externalEvents = new EventEmitter();
2
3 router.get('/contentOwner/disclosure', (req, res) => {
4   ...
```

```
5    externalEvents.emit('session', { code, session});
6  });
7
8  socket.on('connection', (connection: SocketIO.Socket) => {
9    let subscribedCode: string;
10   ...
11   externalEvents.on('session', ({ code, session }) => {
12     if (subscribedCode === code) {
13       connection.emit('session', { session });
14     }
15   });
16 });
```

Listing 3.5: Communication between Express disclosure callback endpoint to Socket.io (simplified)

The validation of the disclosure is mostly done through the uPort and uSocial tools (which we will see in the Integration section). Prior to any matching, we have to make sure that the signatures are correct, especially attestations. uSocial tools makes it easier for us to grab valid attestation data.

While the uSocial platform attests many social networks, we solely utilize email attestations for now. The reasoning behind this is that the InVID Rights Management API is currently matching new accounts with previously existing ones through email, for example, when someone accesses the first time with Google and later with Twitter; and we want to make the uPort integration as seamless as possible with our current infrastructure. InVID Rights Blockchain chooses the best email, according to user email preferences and attested emails and proceeds to do the sign in.

3.7 InVID Reuse Request

The authentication proxy we build with Node allowed us to identify DIDs, through a valid attestation email. The InVID Rights Management server was then able to do the mapping between an existing user or create a brand new account for them.

Users now *feel* like they signed in through uPort because that is how they eventually reached the signed up status, just like when they log in through any other social network. However, there is a catch. When we are signing in through Facebook, Google or Twitter we can acquire a long-lived token that we can use anytime to retrieve updated information about their account, but that is not the same with uPort, on which we need to reask for permission to receive updated information or do any further action on their behalf.

As a side note, most websites that offer authentication through social networks do not need the long-lived token, since they are solely using the social network information to generate a full-fledged account with valid and verified data.

Another difficulty that blockchain with centralized server-side functionality presents is data synchronization and validation. The blockchain interaction that attains the user is meant to be client-side only. The user should be fully able to check the operations they are performing on Ethereum (or even IPFS), and their private keys should never hit the centralized API. That means that if the centralized solution wants to keep itself synced with the blockchain status it has to poll for updates.

In InVID we considered a second option to perform reuse requests over blockchain. The centralized part works the same way as it originally did, actions performed are stored in the database, but we added, optionally, the possibility the execute the same action on Ethereum (through transactions on the smart contract) which shows up as soon as the action has been registered on the API.

The major difficulty of such system is that it is impossible for InVID to have the certainty that Ethereum action is always being replicated by users as soon as the action is stored on the API, as well as the content pushed on the smart contract. During the negotiation procedure, the Ethereum contract might not always be updated by the users, and also some malicious journalist may tweak the smart contract data to fit their needs best. These would not be reflected in the InVID API. We will get into more details about this when we talk about the Solidity smart contract.

If InVID reuse requests only lived in the Ethereum smart contract, we would be able to display the current state of the smart contract through UI. Hence, modifications would always be visible to the user.

A solution to this is either displaying both sides, the API data and blockchain data or make sure the side not displayed through UI is equal to the other before allowing the user to proceed. Given that the blockchain system will work on top of the original one, we will display the API content instead of the blockchain. The client-side will validate the smart contract data against it. We will get into more details in the creation of the reuse request.

3.7.1 Smart contract implementation

Ethereum smart contracts make it possible to store data on the decentralized network, identified and accessible by a contract address. The contracts can have state and logic, which makes it possible for us to store reuse request information as well as to perform some validation.

At InVID we use smart contracts to store the whole trace of the reuse request, from its creation to a possible revocation by the journalist. The data stored under the Ethereum smart contract is the following:

- Reuse request metadata: journalist, content owner and whether it has been revoked.
- Reuse request steps with their terms and metadata, and the parties that have signed that specific step.

```
1  struct ReuseRequest {
2      address journalist;
3      address contentOwner;
4      Step[] steps;
5      uint8 stepsCount;
6      bool revoked;
7      string hash; // IPFS hash
8  }
9
10 struct Step {
11     bytes32 id;
12     bool signedJournalist;
13     uint64 signedJournalistAt;
14     bool signedContentOwner;
15     uint64 signedContentOwnerAt;
16     string hash; // IPFS hash
17 }
18
19 // <reuse request id> -> ReuseRequest
20 mapping (bytes32 => ReuseRequest) public reuseRequests;
```

Listing 3.6: Invid.sol data structures

For each of the steps, it is critical that we store the terms involved in the smart contract. The smart contract by itself should serve as proof that the reuse request occurred at a point in time. While we could associate each of the steps to the unique identifier given by the InVID API, that would make the smart contract rely on the API data, which would make blockchain pointless.

To store the various terms and conditions a reuse request may have we are using IPFS. These can be quite lengthy, which would make transactions very costly for journalists. IPFS makes storing arbitrary data a lot cheaper, and it is also immutable (only one hash can point to a certain file), which is a key property for us to make sure that the information cannot have been tampered with. On the smart contract, we will solely keep a copy of the IPFS hash.

One of the duties of the smart contract logic is to validate that only the involved

parties in the reuse request can take part in the different reuse request actions. The reuse request has no means to contact InVID API to figure out the reuse request participants, so we have to do this in a different way, involving client-side validation.

The journalist will initially provide the content owner address, but also a copy of their disclosureJwt and the content owners. Such token was obtained earlier from the authentication and can be obtained by journalists by querying the API. The disclosureJwt is also uploaded on IPFS as it can be quite lengthy, and it makes it possible to proof the identity of both parties involved (as it contains both a DID and address) utilitzing blockchain only. The address of the content owner passed restricts the people that can take actions over the reuse request on the smart contract. Note that smart contracts work at address level, not DID. We will get into more details about the verification of the identity in the creation of the reuse request.

The logic on the smart contract takes cares of the following:

- Only involved parties can create new steps on the smart contract.
- No more steps can be created after a both parties have signed the final reuse terms.
- The reuse request itself cannot be replaced, reset or modified.

A notable design decision we did was adding the step hash to the signature of the step, looking like the following:

```
1 function signStep(bytes32 _reuseRequestId, string _hash) public
    onlyReuseRequestParticipant(_reuseRequestId) {
```

Blockchain transactions average 20 seconds as of February 2019[24]. A transaction might not be included in a certain block, or two overlapping transactions may be included in the same block (similar to the double-spend attack[25]). A journalist/-content owner could push new reuse terms at the same time as the other party is signing them. With the reuse request ID alone the other party would be signing terms they would have never seen before.

The addition of the hash forces the contract to verify that the latest step hash matches the passed hash, hence signing exactly what they pretend. Alternatively, we could be passing the step identifier, but this makes it more complicated for the web client as the step identifier might not always coincide with the InVID API due to the API having no control over what is uploaded on the blockchain. Both solutions take some more gas to execute, but it is worth it security wise.

Smart contract deployment

The smart contract deployment is handled by Truffle, which deploys the Migration and Invid smart contracts on the specified Ethereum network.

Smart contracts are immutable, no one can make modifications on the logic of the code. However, the Gateway pattern we introduced in the State of the Art does make it possible to switch smart contract through a parent smart contract which address does never change.

For InVID, we believe that the solution to upgrades is a lot simpler. The process of a reuse request may last up to a few days, but it has a clear start and end. Each reuse request can have a specific contract address (or version) on which the process took place, that the users can take note of. Old reuse requests will persist in their original smart contract, and can still be checked anytime by the users. New reuse requests will make use of the newest, which may have slightly different API but similar functionality.

Creation of the reuse request

The remaining part of the blockchain integration is the InVID's client-side for reuse requests. For this part, we will mostly make use of uPort Web3, which is basically a Web3 instance with uPort as a custom provider. We will also be using IPFS Http Client to upload disclosure and step data on Infura's infrastructure.

We want the blockchain modules to be completely isolated from the rest of the web modules, for two reasons:

- The centralized solution will still be the most convenient for InVID users, and we expect most journalists to stick that one.
- The blockchain tools combined weight 6MB uncompressed (750KB compressed and GZIPped).

Thus, it makes no sense to load 6MB modules containing cryptographic browser-based libraries if they are going to use the centralized solution only. For this reason, we are lazy loading modules on demand, which will be downloaded as soon as a user performs an action which requires any of these modules.

```
1 async function lazyLoadModule() {
2   return {
3     uportConnect: (await import(/* webpackChunkName: "ethereum"
        */ 'uport-connect')),
4     uportCredentials: (await import(/* webpackChunkName: "
        ethereum" */ 'uport-credentials')),
5     registerResolver: (await import(/* webpackChunkName: "
        ethereum" */ 'ethr-did-resolver')).default,
```

```
6       invidDidJwt: (await import(/* webpackChunkName: "ethereum"
            */ 'invid-did-jwt')),
7       Web3: (await import(/* webpackChunkName: "ethereum" */ 'web3
            ')).default,
8       web3Utils: (await import(/* webpackChunkName: "ethereum" */
            'web3-utils')).default,
9       ipfsClient: (await import(/* webpackChunkName: "ethereum" */
             'ipfs-http-client')).default,
10      Cryptr: (await import(/* webpackChunkName: "ethereum" */ '
            cryptr')).default,
11      invidContract: (await import(/* webpackChunkName: "ethereum"
             */ '../../../assets/contracts/Invid.json')),
12    };
13  }
```

Listing 3.7: Lazy loading blockchain modules implementation in InVID Rights Management

The first transaction, to create the reuse request on the smart contract, is the one that takes the most gas and the most time to complete since two files have to be uploaded on IPFS (the reuse request one and the first step). As of February 2019, it takes around 0.000287785 Ether ($0.047). It involves defining the reuse request parties, the disclosure data, and the first step metadata and the first step terms.

Since creating the reuse request on the smart contract requires specifying the content owner address (and the journalist, already known when transacting) the UI option, shown in figure 3.15, will only trigger if both parties have previously authenticated through uPort.

The API is currently storing the DID, main Ethereum address and disclosure JWT, and is accessible to the registered journalists on the platform. By using such information the journalist can proceed to execute the first transaction.

From that point, the web client can check whether a reuse request lives on the smart contract by querying a node. If the node returns an existing journalist address different than `0x000`, the reuse request exists.

It is worth mentioning that Solidity smart contract primitives or structs never return null. Instead, they return the default value assigned. For example, a 0 in case of integers or the default values for each of the primitives inside a struct. For this reason, if we want to check whether a reuse request exists we have to pick a value we are sure it will always be set to a different value than its default.

While this option works, and it was, in fact, our first version, that forces the web

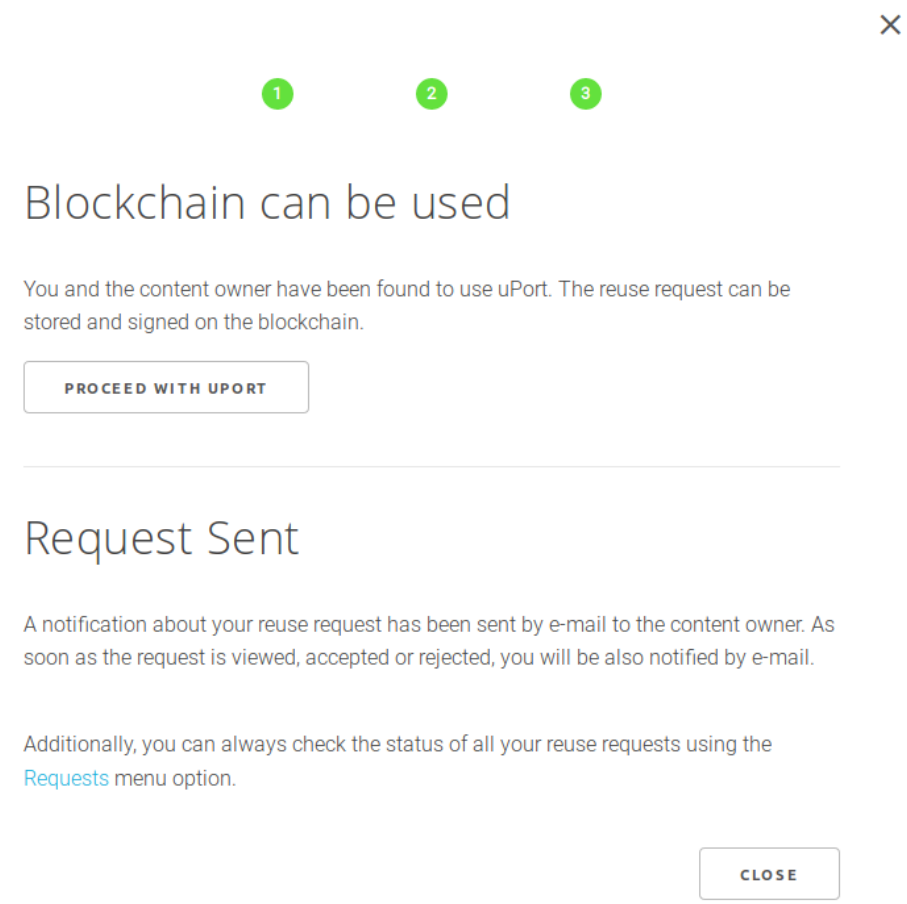


Figure 3.15: Blockchain option displayed to the journalist after requesting for video rights

client to use the Web3 library to check for the existence of such, which is exactly what we were trying to avoid by isolating the modules. The easiest fix is to simply store on the API that we are using Ethereum to handle such reuse request, which will trigger then trigger the web client to do such verification. Alternatively, the API could do the query by itself and return whether it exists, but speed wise the previous solution is better and the fact of the journalist indicating the API whether to use blockchain for the reuse request does not make it less secure.

3.7.2 Signing a reuse request

Once a reuse request is created the content owner is notified. The presented reuse request page will contain the video(s) over which the journalist or their organization request rights and the exact conditions (see figure 3.16).

At first, no blockchain actions are presented. The content owner can agree on the requested terms just like before. Once the reuse agreement has been accepted on

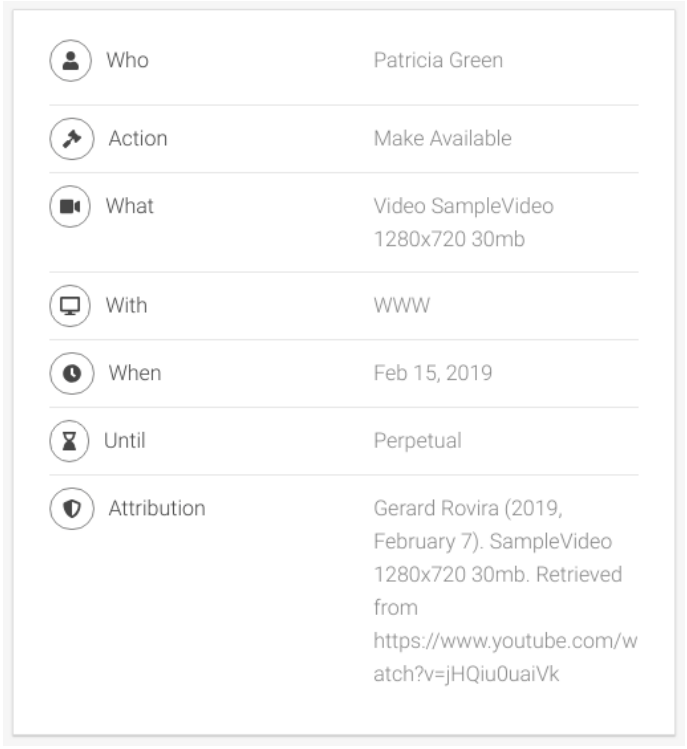


Figure 3.16: Sample reuse request terms

the blockchain we start with the blockchain verification.

Prior we offer the user the possibility to sign the reuse request, the client will make sure the smart contract information is alright, so that the content owner is signing exactly what is displayed on the screen and the journalist address belongs to them (see diagram 3.13 on how we can trace back to the real identity). At this point, the journalist already has the certainty that the smart contract data is correct since they were the ones to create it. The verifications include:

Address verification

To trace back to the real owners behind each of the addresses participating in the smart contract reuse request, we are using the disclosure JWT explained earlier, that is signed by each of the members and includes both the DID and the address.

The first verification step consists of making sure that the addresses in the disclosures (the journalist and content owner addresses) match the participants in the smart contract.

The smart contract address could also be checked against the InVID API. However, the user address may change at some point in the future. When checked against disclosure JWT we rely on the address used for that reuse request. Either way, we do not support changing an address during the process of a reuse request, but we are requesting for the main Ethereum address in the network during the disclosure

request and when doing transactions of any kind, which is technically impossible to change given the current UI tools on the uPort application.

Moreover, the disclosure JWT does also have to be verified. By using `uport-credentials` and `did-jwt` we can verify the JWT signature as well the JWT properties. Most of over verification is exactly the same as the verification of the disclosure, so uPort Credentials does the work for us, except for two properties:

A disclosure request JWT does have an `audience` field. The disclosure JWT was done to the InVID Rights Blockchain server, which was responsible for storing it into the InVID API. Hence, we have to make sure that the audience is, in fact, the DID belonging to the Rights Blockchain instead of the current web client user.

The other is the expiration date. A disclosure request JWT is set to expire within 24 hours. We have to ignore such disclosure expiration time because the disclosures will be stored perpetually on IPFS, and they will never be updated, but we will want them to be valid when checking them in the future. Alternatively, we could check whether it was valid at a certain point in time, but the implementation is fuzzier since the disclosures are always sent at an earlier point in time rather than when a reuse request is being created and we would have to rely on the API to store the disclosure times.

DID verification

The address verification was one half of the identity verification, the other half involves verifying the DID. We want to make sure that they do match the ones stored in the InVID API. If so, any user can eventually trace back to the video owner with the help of the social media platform and the information stored in the smart contract.

The smart contract DID will also have to be obtained through the disclosures on IPFS, performing the same verification over the JWTs.

Verify step content

Finally, we have to verify whether the reuse terms on the smart contract match exactly the ones being displayed on the screen.

To do so, we are generating a JSON version of the terms, including video ownership from the API and we are checking the stored version on the IPFS against it. If both are identical, the last step uploaded by the other party was correct and the other party can proceed with the signature.

There is one problem with such verification system though. Small modification to the video metadata or content owner information will make such verification fail. To

fix it, we can reduce the data stored on IPFS, or make the API be the one generating such version. This will the JSON will always be fixed to certain values, instead of dynamically changing.

When two validation does not succeed, we provide the user the option to re-upload the step, just like we will explain next in the terms negotiation. It will then be the other party the responsible for signing it.

3.7.3 Negotiating terms

The ideal reuse request will start by the journalist request some rights over media, and the content owner accepting/signing such reuse request without further hesitating, meaning only one reuse request step is needed. However, there are times when the content owner disagrees with some of the terms and wants to propose new more restrictive ones. That involves creating further steps.

Our blockchain implementation aims to mimic the steps stored in the API, keeping a copy of each of the step terms until both parties agree on a set of terms.

There are a couple of ways to implement such a system on top of the existing one:

- Prompt to upload the step as soon as new terms have been requested.
- Verify whether the last stored step on the blockchain matches the latest one in the API (also shown through UI).

The first solution would be similar to what we are already doing to create new reuse request, on which we are triggering an IPFS and Web3.js actions as soon as the API has been updated.

While there is nothing particularly wrong with that solution, we still have to validate the Ethereum step before the final signature from the other party. Hence, by executing a verification already after negotiating terms we are already implementing the validation mechanism that we need.

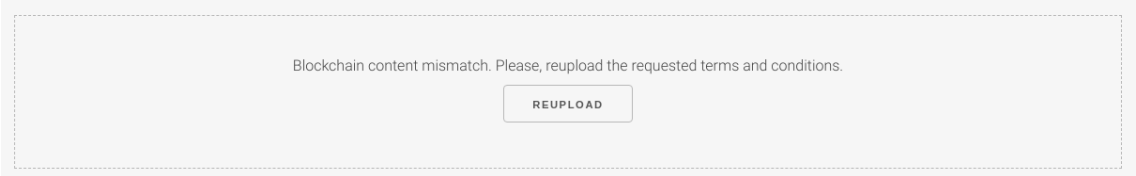


Figure 3.17: Blockchain verification failure after requesting for new rights

3.7.4 Revoking terms

One of the stakeholder demands was the ability to revoke reuse request that had been granted from content owners to journalists. That is only possible one-way, from the journalist side, the one who will make use of the requested rights. Content owners are given a journalist contact address that they can use to describe why they believe the reuse request should be revoked.

Such has to be reflected on Ethereum as well if they are using blockchain to handle the reuse request.

As we have seen before, our reuse request structure carries a boolean that indicates whether the reuse request has been revoked. Revoking it on the web will once again, pop an alert indicating that such change such has to be reproduced on the blockchain for it to be complete.

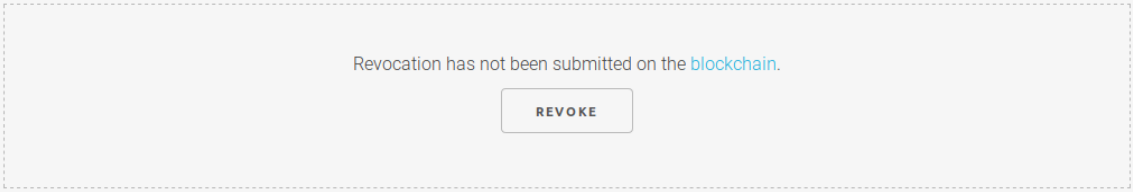


Figure 3.18: Revocation has to be submitted on the blockchain UI message

3.7.5 Blockchain as a backup system

We use blockchain as a secondary system, that works along with the original API server-side solution. The use of Ethereum does never replace the API, and the web client uses it to display most of the content that shows on the different pages.

However, blockchain makes our system more reliable, as it is tamper-proof, more resilient to downtime, and anyone can check the exact logic that is being run behind the scenes.

For this reason, InVID users should be able to save themselves a reference to the Ethereum smart contract and its reuse request, so that they can check its status at any point without the need to access the InVID website as it is a completely isolated service, which is what makes it decentralized.

The information that users need to be able to do so is the following:

- Ethereum network: The default Ethereum network should be mainnet. Although, given that uPort support to mainnet is still under development, we are

working with Rinkeby testnet. Either way, the network is a constant variable.

- IPFS network: Just like Ethereum network, IPFS is a constant that described where the files associated with Ethereum smart contract reuse requests reside. We are using Infura IFPS.
- Contract address: This is the first key value to locate where the reuse request were stored. There are thousands of smart contracts published on the Ethereum network and without the contract address, it would be impossible for a user to find them.
- Reuse request ID: InVID Rights Management may have processed a few reuse requests, which could make it slightly difficult to locate a particular smart contract. For this reason, we are providing the reuse request ID so that they can access it directly. Alternatively, by using Web3.js and the provided smart contract events we could filter the ones belonging a particular journalist/content owner or date.
- Encryption key: The encryption key is associated to each reuse request and makes it possible to read and understand the content stored on IPFS. We will get into more details next.

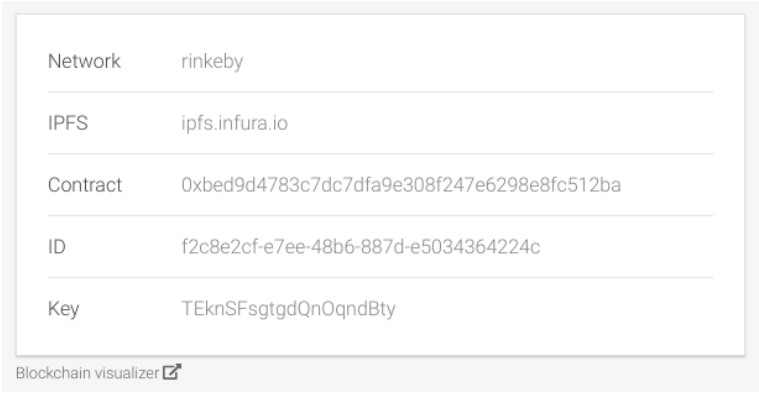


Figure 3.19: Blockchain information table presented on each reuse request page

3.7.6 Encrypting personal data

One of the concerns, when we are working with decentralized data, is the privacy behind the data that is uploaded on the decentralized network. Ethereum nodes keep a copy of every single transaction, which includes smart contract transactions with the associated data. That makes it possible for anyone to read the state of the smart contract.

We believe that a reuse request should be kept private between the journalist and

content owner (the parties involved), and should only be accessible with their express consent. Furthermore, our smart contracts are storing disclosure requests, personally identifiable data that makes it possible to match their Ethereum pseudo anonymous identity with their official one.

What they both have in common is that they are stored in IPFS, so we recognize that the privacy layer has to be applied over IPFS content.

An encryption algorithm such as AES[26] makes it possible to store such content while making it possible for the private key keepers to read it afterward.

For this, we can use the API to generate a random key for each of the requests that are created on the site. Since only the journalist and content owner have access to the API stored reuse request, they are the only one who will have access to the private key. If they ever have to access to the blockchain content, they will have to use that key (displayed on the UI as shown earlier), to decipher the IPFS content.

The encryption key is also part of the validation process. If the IPFS content is illegible through the API provided means, it means that the other party uploaded the content with another key.

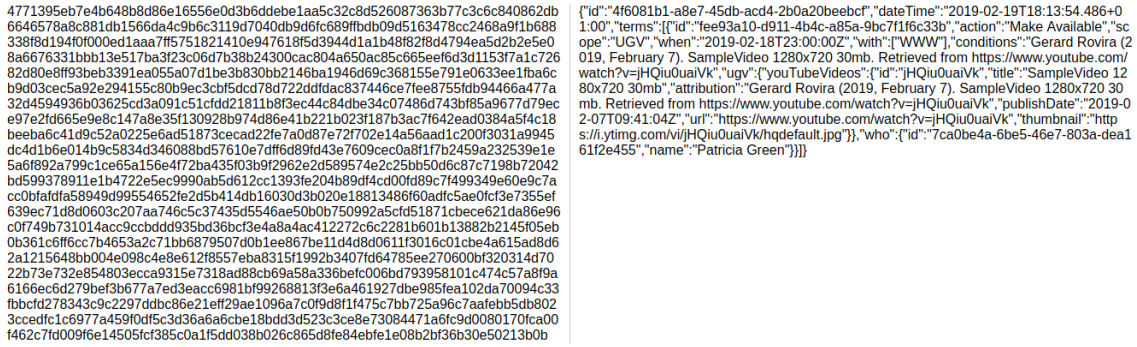


Figure 3.20: Encrypted on the left, decrypted on the right

In figure 3.20, we show file with hash `Qmf7KBtcLZEqMAuoUpXRDwBR3aejG WUoThaKwpiNrq2yeW` stored in Infura IPFS (`https://ipfs.infura.io/ipfs/ Qmf7KBtcLZEqMAuoUpXRDwBR3aejGWUoThaKwpiNrq2yeW`) with key `TEknSFsgtgdQn OqndBty`.

3.8 InVID Blockchain Visualizer

In the previous section, we described the implementation decisions behind the blockchain in InVID Rights Management. At this point, a journalist can request

the use the of uPort to store reuse request on Ethereum and IPFS.

However, we believe it is difficult for users to verify themselves whether data is being stored correctly in the blockchain, and how the data in the blockchain looks like. Web3.js and IPFS HTTP Client or similar tools have to be used to retrieve all the data, and for most cases, it is not time and economically worth it to do their our implementation for that.

For this reason, we are providing our own blockchain visualizer, shown in figure 3.21, on which users can see exactly what is in the blockchain without technical knowledge.

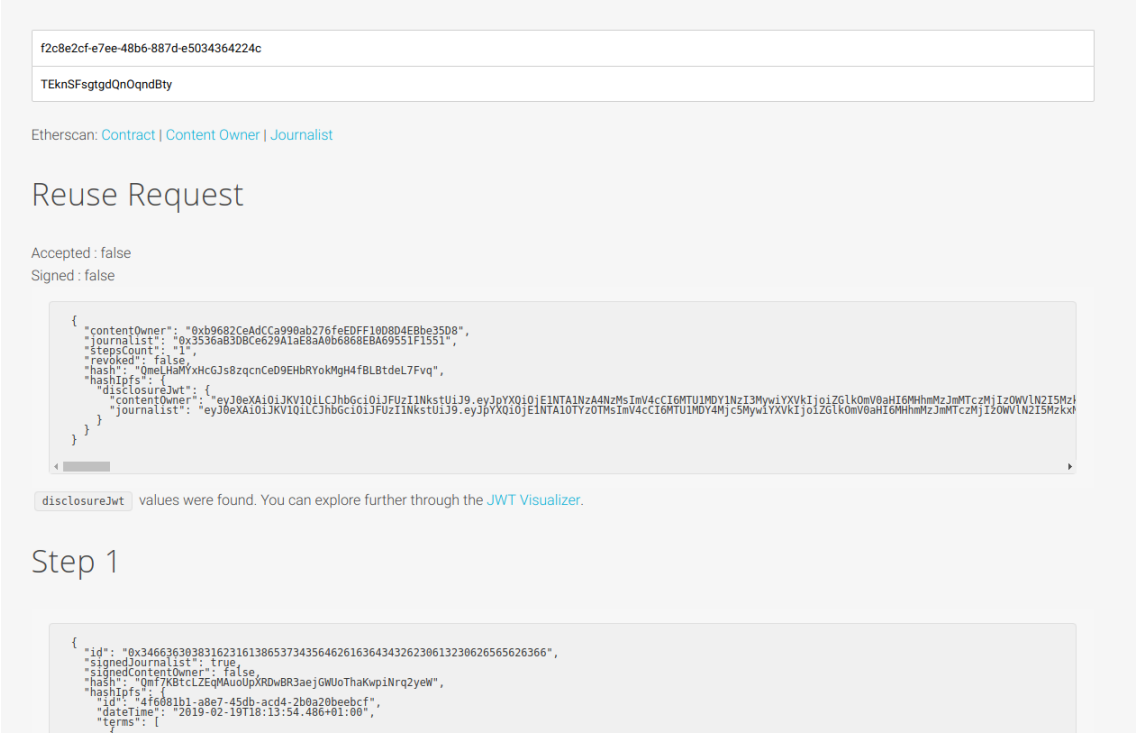


Figure 3.21: InVID blockchain visualizer

The blockchain queries start with Ethereum. With Web3.js and our smart contract ABI, we are able to call our smart contract methods that will return the stored data. Solidity automatically generates getter methods to retrieve contract properties, which we can use to fetch those that do not have a method associated.

The first query will consist of retrieving the reuse request metadata stored, then we will make use of this ID to retrieve every single step.

The remaining data is stored in the IPFS, which we can locate through the hash stored in the smart contract. We will request the decryption token to the user first, then use it to attempt to the decryption of the content.

For the disclosure requests, users who have the key will be able to read the information that associates a journalist/content owner address with their uPort DID and email address. The DID nor the attested email does give us the certainty that the media belong to the content owner. However, given the case, the media provider could help demonstrate that the real owner of the video matches the given email address.

3.9 Integration

When we defined uSocial, we had to key features in mind:

- Serve as the point of trust that will verify the various social accounts.
- Provide an API for developers to rely on to make use of these verifications.

What we have been doing with InVID Blockchain is exactly the integration we are talking about, except that it should be straightforward to determine valid attestations, most current or simply retrieve valid values. With InVID Blockchain the verification was written was from scratch, which is not what we are expecting developers to do.

The integration step consists in providing a Node.js compatible library for developers to use our service without hassle, under the name *usocial*. The package has been uploaded onto NPM, the most popular JavaScript registry.

So what exactly is that missing library about?

uPort does a good job verifying disclosure request through uPort Connect, which includes linked attestations. This verification includes JWT specifics, such as signature or expiration date.

uPort can also verify an attestation JWT directly and extract its payload through uPort Credentials.

However, a uPort attestation consists of arbitrary data packed in by the creator of the attestation. With the correct private keys belonging to the signer DID, anyone can build an attestation that follows a certain format and name it as they like. That means, that anyone can mimic Usocial Identity attestations.

The usocial library will take care of verifying the attestation format, the issuer and receiver so that other developers do not have to worry about validation. uPort should still be used to do the validating.

One of the design decisions that we faced prior getting started with this was whether to include the JWT verification with the library so that instead of being a completely separate module, it worked on top of uPort. But we think it is less flexible this way, uPort provides at least 3 different ways to process JWT tokens, with many customizable parameters. Working on top of uPort would certainly remove that flexibility, which we were unsure for now. All and all, both options can still be supported in the future.

```
1  [...]
2  import { verifyAttestation } from 'usocial';
3
4  [...]
5  // uport-credentials authenticate disclosure
6  const profile = await credentials.authenticateDisclosureResponse
       (jwtToken);
7
8  // usocial verify attestation (format, issuer and subject)
9  if (!verifyAttestation(profile.verified[0], { sub: profile.did,
       iss: USOCIAL_DID })) {
10   throw new Error('Attestation is not valid');
11 }
```

Listing 3.8: Verify attestations with usocial library

Having the tools to figure out correct attestations, we can facilitate attested values to the user, as well as the most current valid attestation.

For services such as Rights Blockchain, or even our own uSocial API and client we can rewrite our previous implementation to use usocial.

```
1  [...]
2  import { attestedEmails } from 'usocial';
3
4  [...]
5  const profile = await credentials.authenticateDisclosureResponse
       (accessToken);
6
7  const verifiedEmails = attestedEmails(profile.verified, { sub:
       profile.did, iss: USOCIAL_DID });
8  let email: string;
9  if (verifiedEmails.includes(profile.email)) {
10   email = profile.email;
11 } else if (verifiedEmails.length > 0) {
12   email = verifiedEmails[0];
13 } else {
14   email = null;
```

```
15 }
```

Listing 3.9: InVID Blockchain email selection implementation from uPort authentication disclosure

Chapter 4

Conclusions

4.1 Conclusions

The development of this project has given us a greater understanding of what blockchain is about, how it works and some of its use cases. In particular, we have focused on Ethereum.

Ethereum is still in the early ages, but it has a lot of potential, especially for developers like us who want to use the blockchain as a timestamp proof or/and execute logic funds under certain conditions. Ethereum makes the distribution of funds automatic and transparent, as accounts and smart contracts reside in every single network node.

We also have to acknowledge the huge community projects that have grown around Ethereum during the past 3 years since it was released that have made it possible for us to build uSocial and InVID Rights Management integration within few months.

To start, uPort provided us the foundation of our project. A system to create and manage identities on top of Ethereum. Every uPort user is assigned a DID address and can have one of Ethereum addresses, just like a wallet. uPort also provides an attestation system, on which arbitrary data signed by other parties can be stored on the users' own device. What made the uPort project features worthy for us are the APIs that made it possible to interact with its data programmatically: uport-connect, uport-credentials, did-jwt, ...

uSocial was built around uPort, given that the identity and attestation system is exactly what we needed to get the verification of identities project running.

uPort, other than the transports that they use to communicate between services

and optional backup systems, is fully decentralized. Most of the data reside in each of the user devices, and some in an Ethereum smart contract. That makes it fully transparent and gives users the security that they are under control of their own data. This is reinforced by the fact that their mobile application also displays every single bit of personal information being requested by third-party services prior sharing.

During the development of the project, we also utilized MetaMask to test and compare two different providers with our smart contract. MetaMask is one of the most popular wallets available. This one gives the users a lot more control over each of the accounts. Although it did work flawlessly for transactions, it does not count with the identity system that uPort has, which is currently a requirement for our project.

The truffle framework, by Consensys, makes it straightforward to get started with Solity smart contracts on any development environment and gives the necessary tools to easily integrate it with Continuous Integration systems. We had tried Remix IDE in the past, but we felt the development, from folder structure system to deployment, even though it was easier to understand it was too tight to their IDE. Truffle has made it possible for us to integrate testing and deployment with NPM in our InVID Rights Blockchain project built with TypeScript. This also makes it possible to easily integrate Solity tests with the existing CI.

Truffle also makes it possible to simulate an Ethereum test network, either through their own test framework or Ganache (also part of the Truffle Suite). This made it possible for us to have a journalist, a content owner and a guest account to test the various roles with each other and make sure the contract behaves as expected. When working with smart contracts, testing thoroughly is critical since there is no possibility for hotfixes unless you do a smart contract hot-swapping through the Gateway contract, but still not ideal. While the InVID smart contract does not involve funds, it is still extremely important that only the proper identities perform actions over the submitted reuse requests, otherwise, it defeats the purpose of the smart contract. Moreover, it is difficult to debug a deployed smart contract through existing tools.

Web3.js by Ethereum, the JavaScript-based tool which we used both directly and through uPort, does the RPC protocol implementation for communicating with an Ethereum tool and makes it easy to push modifications to the contract and also to read existing data.

Other than the ones described earlier, the Ethereum ecosystem counts with many more applications, which is what makes Ethereum really appealing for businesses

wanting to integrate it with their own existing solutions.

On the other hand, the need for larger storage to store each of the reuse request steps in InVID and identity proofs gave us the opportunity to learn and work with IPFS. It takes the well-known P2P networks to the next level. Files are identified by a unique fingerprint, which makes files content tamper-proof. This made IPFS a great companion to the InVID smart contract.

What we believe the biggest IPFS downside for us is the fact that files can eventually be *abandoned*. IPFS power relies on peers serving the copies of the files, but each of the nodes may just serve a small subset of the files. For the time being, we are relying on Infura IPFS, which is backed by their own nodes. Later on, InVID may have to add their own nodes to pin such data to circumvent that. Related, IPFS can have bandwidth problems.

With Ethereum, data also relies on peers. However, every single network has a copy of all the data in the network (at least while a sharding solution is not active), and the current Ethereum network size is big enough to prevent that from happening.

All in all, this project has been very educative and given that the scope of the project was working with Ethereum on the web, it has also helped us acquire a more solid grounding on React and Angular client-side library/framework as well as Node for the back-end. The uSocial platform complies with our initial expectations, and it would now be ideal that trustworthy entities considered this working prototype as an example of what to work on to get the most out of the decentralized web.

InVID Rights Management does now support blockchain to handle reuse requests. Journalists who are curious about blockchain or want to take advantage of its benefits can choose to use it. However, given that the centralized solution is still the most comfortable for both parties and involves the least steps (especially because the supported Facebook, Twitter and YouTube media sources are centralized), we expect most to stick with the traditional method. It is very common for people to have a Google account, while it is rare to find users having a uPort account.

Chapter 5

Future Work

5.1 Future Work

Overall, the project does what it was expected to do, but there is room for features and enhancements. To list some:

UX

Ideally, a decentralized application should be as easy to use as a traditional one backed with a centralized API. We understand that decentralization often involves new technology, such as using a wallet instead of a traditional bank account, or DID instead of a username and password, that users may have to learn and get used to over the time.

However, neither uSocial nor the InVID integration as a UX friendly as they should be.

On the one hand, uSocial is designed with the concept of user dashboard in mind. We wanted users to feel like they are managing their uPort account on the web, just like when you log in with a social network account. And then the attestation generator was built on top of this idea, marking it as the core feature of the dashboard.

Behind the scenes though, there is no user management. The data is just read-only and the attestations that do not take place in the same browser session are not reflected in their account unless they re-sync. This can be somewhat confusing for newcomers.

A possible solution to this would be to add the possibility the synchronize the web data with the mobile device in real time. To do so, we suggest to implement it on uport-connect as a specific permission, just like pushToken, on which you can

request permission update or simply poll new updates.

On the other hand, InVID presents numerous UX issues, some of which would require a more complex server-side solution:

In InVID, journalists are required to request themselves an account that will be manually validated and approved by a member of the InVID team. Before uPort was implemented, email and password was the only login option for them. With uPort, they now require to previously sign up with the traditional option. A uPort disclosure request can return most of the form information without the need of typing it by hand. Hence, a custom uPort sign up would make more sense.

Secondly, it is unclear how uPort works for InVID as it will only pop up when both parties are using it. We should either clarify how it works during the creation of the reuse request or change how it works completely.

The change we have in mind is quite complex and make significant changes in the current flow so we moved it into the next bullet point which we will cover in more detail.

InVID - reuse request flow

When no uPort is used, journalists can create a reuse request over a random video and the platform will invite them to join InVID. After joining, which only takes logging in with the social platform, they can immediately accept the reuse request, which completes the reuse request flow.

The current uPort implementation is somewhat more cumbersome. Both journalists and content owners are required to have previously signed in with uPort in the platform as well as the media social site where the video is hosted. That makes it difficult for a journalist to start a reuse request throgh uPort.

The current implementation makes the original flow very clear, since it works on top of it, but not that easy to get started it.

We suggest a brand new flow for journalists wanting to use uPort:

First of all, the reuse request can be either carried over the traditional way or Ethereum backed to InVID (see figure 5.1).

When using uPort, journalists will be prompted to a disclosure request if they have never accessed through uPort before. That will make it possible for us to still verify his identity prior to getting started with the reuse request.

The content owner might have never accessed the site before, and we should be counting with it. The smart contract should be modified not to require it at first.

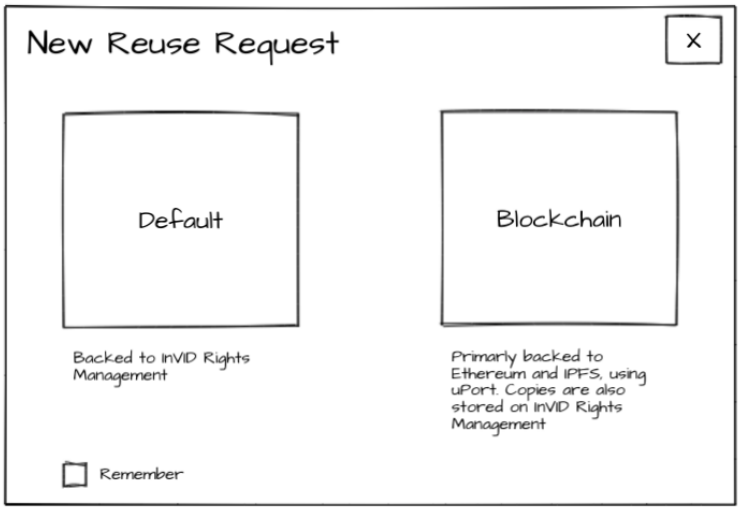


Figure 5.1: Wireframe of a possible new initial reuse request screen

A possible solution would be to accept new steps from arbitrary content owners, which the web client will verify afterward. That means, we would be getting rid of the reuse request contentOwner field and its hash would solely store the journalist one.

The disclosure request from the content owner would be passed every time with the step hash. We recognize that this makes it more difficult to validate, but Ethereum does have no means to determine whether the content owner is rightful or not. Only the web client is able to extract IPFS data and verify the JWT. And anyone with access to an Ethereum node can poll new reuse requests almost in real time and consequently extract their ID and use it for calling smart contract functions.

Contrary to the current reuse request page, which displays InVID information instead of blockchain, and is later verified, the information will be directly printed from the blockchain.

Once a transaction is completed, the web client will inform the API of the changes. Since the API should not trust the web client updates, the API or Rights Blockchain as a proxy, will have to pull Ethereum changes from a network node and update the API database. This way InVID will still have a backup of the data just like with the traditional system.

InVID - use signature instead of transaction

One of the most promising upcoming features of uPort right now is the ability to sign instead of transacting.

What this means for us is that neither the journalist nor content owner will have to have Ether on their account to create the reuse request or submit the steps. The

can simply sign it with their account private key and send it to our API, that will upload on the Ethereum network for them without any cost.

The added difficulty here would only be verifying that the signature responsible for creating a reuse request or step matches the real creator instead of the API.

uSocial - remove attestations

Currently, there is no way to remove an attested piece of data, for example a particular email account. New attested items are added on top of existing ones.

The web client passes the most current attestation on the stateless server, that will verify the JWT, make sure it was verified by it and put the new verification on top.

Being able to remove a particular reuse request would be a good addition, and it would be server-side work as the client cannot edit the attestation to make it shorter in any way, as it is packed in a signed JWT by the API.

The web client should then inform the API of the particular item that it wants to have returned and the API will push a new shorter attestation to the mobile device.

Switch networks

For the time being, uPort seems to have the most support for Rinkeby, and Ethereum mainnet does not seem to be supported for smart contracts.

As soon as the implementation is ready, it would be good to support the mainnet and some testnets on both uSocial and InVID.

MetaMask

Since the very beginning, this project has moved towards uPort for the ability to create decentralized identities and attestations that are so useful for us during the course of the project.

However, there exist many more implementations of wallets that we cannot forget, such as MetaMask, MyEtherWallet[1], or Status[2].

Although we are unsure of a solution to this, it would ideally be something that can work on top of their current implementation, so that we can leverage their current seed to generate a decentralized identifier which we can use to attest information.

[1]MyEtherWallet: `https://www.myetherwallet.com`
[2]Status: `https://dev.status.im`

Other than the ones listed, Ethereum and its ecosystem keep changing rapidly. More and better ways to handle the verification of identities and integration may come up as the existing tools evolve and new tools appear in the market.

Appendix A

InVID Rights Management Smart Contract

Also available at:

https://gist.github.com/zurfyx/0a51fc77322b2f83cf7115f4f4dfae3c

```
1  pragma solidity ^0.4.24;
2
3  import "./Ownable.sol";
4
5  /**
6   * @title Contract to register reuse requests on the InVID Rights
       Management site.
7   * @dev This contract stores the various request requests and their
        parties, as well as each of the
8   * reuse request steps that were taken during the negotiation
       process.
9   * The reuse request terms for each of the steps can be found on
       IPFS.
10  */
11 contract Invid is Ownable {
12
13     struct ReuseRequest {
14         address journalist;
15         address contentOwner;
16         Step[] steps;
17         uint8 stepsCount;
18         bool revoked;
19         string hash; // IPFS hash
20     }
21
22     struct Step {
```

```
23          bytes32 id;
24          bool signedJournalist;
25          uint64 signedJournalistAt;
26          bool signedContentOwner;
27          uint64 signedContentOwnerAt;
28          string hash; // IPFS hash
29      }
30
31      mapping (bytes32 => ReuseRequest) public reuseRequests; // <
            reuse request id> -> ReuseRequest
32
33      event NewReuseRequest(bytes32 indexed reuseRequestId);
34      event NewStep(bytes32 indexed reuseRequestId, uint stepIndex);
35      event SignedStep(bytes32 indexed reuseRequestId, uint stepIndex
            );
36      event RevokedReuseRequest(bytes32 indexed reuseRequestId);
37
38      modifier onlyReuseRequestParticipant(bytes32 _reuseRequestId) {
39          ReuseRequest storage _reuseRequest = reuseRequests[
                _reuseRequestId];
40          require(
41              msg.sender == _reuseRequest.journalist || msg.sender ==
                    _reuseRequest.contentOwner,
42              "Unauthorized (neither journalist nor content owner)"
43          );
44          _;
45      }
46
47      modifier onlyJournalist(bytes32 _reuseRequestId) {
48          ReuseRequest storage _reuseRequest = reuseRequests[
                _reuseRequestId];
49          require(msg.sender == _reuseRequest.journalist, "
                Unauthorized (not journalist)");
50          _;
51      }
52
53      /*
54       * Creates a new reuse request, considering the creator as the
             journalist.
55       * Additionally, the first step is created and marked as signed
              on the journalist side.
56       */
57      function createReuseRequest(
58          address _contentOwner,
59          bytes32 _reuseRequestId,
60          string _reuseRequestHash,
61          bytes32 _firstStepId,
```

```
62          string _firstStepHash
63      ) public {
64          require(_contentOwner != address(0), "Content Owner address
                should not be 0");
65          require(reuseRequests[_reuseRequestId].stepsCount == 0, "
                There already exists a reuse request with that Id");
66
67          reuseRequests[_reuseRequestId].journalist = msg.sender;
68          reuseRequests[_reuseRequestId].contentOwner = _contentOwner
                ;
69          reuseRequests[_reuseRequestId].hash = _reuseRequestHash;
70
71          emit NewReuseRequest(_reuseRequestId);
72
73          createStep(_reuseRequestId, _firstStepId, _firstStepHash);
74      }
75
76      /**
77       * Creates a new step on the reuse request, if:
78       * - Participant
79       * - Both participants haven't agreed on a step (unfinished
             reuse request)
80       * The signature on the creator side will already be set.
81       * Note: Two steps can have the same stepId. Participants will
             still require the step index
82       * to sign, so it doesn't matter.
83       * Note2: Cancelled agreements can't have further steps either.
84       */
85      function createStep(bytes32 _reuseRequestId, bytes32 _stepId,
            string _hash) public onlyReuseRequestParticipant(
            _reuseRequestId) {
86          require(!signedBoth(_reuseRequestId), "Reuse request is
                complete. No further steps allowed");
87
88          ReuseRequest storage _reuseRequest = reuseRequests[
                _reuseRequestId];
89          Step memory step = msg.sender == _reuseRequest.journalist
90              ? createStepJournalist(_stepId, _hash)
91              : createStepContentOwner(_stepId, _hash);
92          _reuseRequest.stepsCount = uint8(_reuseRequest.steps.push(
                step));
93
94          emit NewStep(_reuseRequestId, _reuseRequest.stepsCount - 1)
                ;
95      }
96
```

```
97      function createStepJournalist(bytes32 _stepId, string _hash)
            internal view returns (Step) {
98          return Step({
99              id: _stepId,
100             signedJournalist: true,
101             signedJournalistAt: uint64(block.timestamp),
102             signedContentOwner: false,
103             signedContentOwnerAt: 0,
104             hash: _hash
105         });
106     }
107
108     function createStepContentOwner(bytes32 _stepId, string _hash)
            internal view returns (Step) {
109         return Step({
110             id: _stepId,
111             signedJournalist: false,
112             signedJournalistAt: 0,
113             signedContentOwner: true,
114             signedContentOwnerAt: uint64(block.timestamp),
115             hash: _hash
116         });
117     }
118
119     /*
120      * Signs the latest reuse request step, as long as the hash is
            valid.
121      * This function can be called unlimited times, but following
            times will have no effect.
122      */
123     function signStep(bytes32 _reuseRequestId, string _hash) public
            onlyReuseRequestParticipant(_reuseRequestId) {
124         ReuseRequest storage _reuseRequest = reuseRequests[
                _reuseRequestId];
125         Step storage _lastStep = _reuseRequest.steps[_reuseRequest.
                stepsCount - 1];
126         require(
127             keccak256(abi.encodePacked(_lastStep.hash)) ==
                    keccak256(abi.encodePacked(_hash)),
128             "Latest step hash doesn't match provided"
129         );
130
131         if (msg.sender == _reuseRequest.journalist) {
132             signStepJournalist(_lastStep);
133         } else {
134             signStepContentOwner(_lastStep);
135         }
```

```
136
137         emit SignedStep(_reuseRequestId, _reuseRequest.stepsCount -
                1);
138     }
139
140     function signStepJournalist(Step storage step) internal {
141         step.signedJournalist = true;
142         step.signedJournalistAt = uint64(block.timestamp);
143     }
144
145     function signStepContentOwner(Step storage step) internal {
146         step.signedContentOwner = true;
147         step.signedContentOwnerAt = uint64(block.timestamp);
148     }
149
150     /**
151      * Whether the last reuse request step has been signed by all
            parties (journalist & content
152      * owner).
153      * Beware: the reuse request may have been revoked. Use
            isAccepted() instead to find out whether
154      * the reuse request still prevails.
155      */
156     function signedBoth(bytes32 _reuseRequestId) public view
           returns(bool) {
157         ReuseRequest memory _reuseRequest = reuseRequests[
                _reuseRequestId];
158         if (_reuseRequest.stepsCount == 0) {
159             return false;
160         }
161         uint _lastIndex = _reuseRequest.stepsCount - 1;
162         Step memory _lastStep = reuseRequests[_reuseRequestId].
                steps[_lastIndex];
163         return _lastStep.signedJournalist && _lastStep.
                signedContentOwner;
164     }
165
166     /**
167      * Whether as a journalist, you can make use of the reuse
            request terms.
168      * This function takes into consideration the revoked status
            stored on the reuse request.
169      */
170     function isAccepted(bytes32 _reuseRequestId) public view
           returns(bool) {
171         return signedBoth(_reuseRequestId) && !reuseRequests[
                _reuseRequestId].revoked;
```

```
172     }
173
174     function getReuseRequestStep(bytes32 _reuseRequestId, uint
            stepIndex) public view
175         returns(bytes32 id, bool signedJournalist, bool
                signedContentOwner, string hash) {
176
177         Step memory step = reuseRequests[_reuseRequestId].steps[
                stepIndex];
178         return (
179             step.id,
180             step.signedJournalist,
181             step.signedContentOwner,
182             step.hash
183         );
184     }
185
186     /**
187      * As a journalist, you can revoke previously accepted reuse
             request.
188      * This function can be called unlimited times, but following
             times will have no effect.
189      */
190     function revoke(bytes32 _reuseRequestId) public onlyJournalist(
            _reuseRequestId) {
191         require(signedBoth(_reuseRequestId), "A reuse request
                cannot be revoked until both parties have signed");
192
193         ReuseRequest storage _reuseRequest = reuseRequests[
                _reuseRequestId];
194         _reuseRequest.revoked = true;
195
196         emit RevokedReuseRequest(_reuseRequestId);
197     }
198 }
```

Listing A.1: Invid.sol data structures

Bibliography

[1] M. Anicas, "An introduction to oauth 2." https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2, July 2014. [Online; accessed 14 January 2019].

[2] Salesforce, "Oauth 1.0.a authentication flow." https://help.salesforce.com/articleView?id=remoteaccess_oauth_1_flows.htm&type=5. [Online; accessed 15 January 2019].

[3] E. R. P. L. e. E. B. E. H. Kai Wagner Jolocom, Balázs Némethi Taqanu, "Self-sovereign identity." https://www.bundesblock.de/wp-content/uploads/2019/01/ssi-paper.pdf, October 2018. [Online; accessed 05 February 2019].

[4] C. Team, "Decentralized networks." https://medium.com/coinbundle/decentralized-networks-1d5e2e92953b, August 2018. [Online; accessed 05 February 2019].

[5] J. S., "Blockchain: how a 51% attack works (double spend attack)." https://medium.com/coinmonks/what-is-a-51-attack-or-double-spend-attack-aa108db63474, May 2018. [Online; accessed 05 February 2019].

[6] "The ethereum-blockchain size has exceeded 1tb, and yes, it's an issue." https://hackernoon.com/the-ethereum-blockchain-size-has-exceeded-1tb-and-yes-its-an-issue-2b650b5f4f62, May 2018. [Online; accessed 24 February 2019].

[7] R. Jordan, "How to scale ethereum: Sharding explained." https://medium.com/prysmatic-labs/how-to-scale-ethereum-sharding-explained-ba2e283b7fce, January 2018. [Online; accessed 05 February 2019].

[8] D. G. Wood, "Ethereum: A secure decentralised generalised transaction ledger." http://gavwood.com/paper.pdf, March 2017. [Online; accessed 26 February 2019].

[9] S. Khatwani, "9 anonymous cryptocurrencies you should know about." https://coinsutra.com/anonymous-cryptocurrencies/. [Online; accessed 04 February 2019].

[10] B. Vitaris, "Swiss crypto valley to create digital identities for its citizens on the ethereum blockchain." https://bitcoinmagazine.com/articles/swiss-crypto-valley-create-digital-identities-its-citizens-ethereum-blockchain/, July 2017. [Online; accessed 04 February 2019].

[11] P. Braendgaard, "Different approaches to ethereum identity standards." https://medium.com/uport/different-approaches-to-ethereum-identity-standards-a09488347c87, January 2018. [Online; accessed 20 December 2018].

[12] "Introduction to json web tokens." https://jwt.io/introduction. [Online; accessed 26 February 2019].

[13] uPort, "Attesting credentials." https://developer.uport.me/guides/attestcredentials. [Online; accessed 05 February 2019].

[14] D. L. C. A. R. G. M. S. Drummond Reed, Manu Sporny, "W3c decentralized identifiers (dids) v0.11." https://w3c-ccg.github.io/did-spec/, January 2019. [Online; accessed 27 December 2018].

[15] "Javascript api." https://github.com/ethereum/wiki/wiki/JavaScript-API, January 2019. [Online; accessed 15 December 2018].

[16] M. Belenkiy, "Designing an upgradeable ethereum contract." https://medium.com/centre-blog/designing-an-upgradeable-ethereum-contract-3d850f637794, September 2018. [Online; accessed 24 December 2018].

[17] uPort, "Myetherwallet behind-the-scenes." https://kb.myetherwallet.com/transactions/transactions-not-showing-or-pending.html. [Online; accessed 05 February 2019].

[18] "A next-generation smart contract and decentralized application platform." https://github.com/ethereum/wiki/wiki/White-Paper, August 2018. [Online; accessed 27 December 2018].

[19] G. Rushgrove, "User stories for web operations teams." https://www.morethanseven.net/2017/01/01/user-stories-for-web-operations/, January 2017. [Online; accessed 10 January 2019].

[20] V. Driessen, "A successful git branching model." https://nvie.com/posts/a-successful-git-branching-model/, January 2010. [Online; accessed 7 January 2019].

[21] T. F. S. Foundation, "Daily use of gnupg." https://www.gnupg.org/gph/en/manual/c481.html, 1999. [Online; accessed 18 January 2019].

[22] Auth0, "Oauth 2.0 state parameter." https://auth0.com/docs/protocols/oauth2/oauth-state. [Online; accessed 18 January 2019].

[23] Twitter, "Post oauth/request_token." https://developer.twitter.com/en/docs/basics/authentication/api-reference/request_token. [Online; accessed 20 January 2019].

[24] Etherscan, "Ethereum average blocktime chart." https://etherscan.io/chart/blocktime. [Online; accessed 15 February 2019].

[25] B. Project, "Double spend." https://bitcoin.org/en/glossary/double-spend. [Online; accessed 15 February 2019].

[26] "Advanced encryption standard (aes)." https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf, November 2001. [Online; accessed 19 February 2019].